DEMONSTRATION OF INEXACT COMPUTING IMPLEMENTED IN THE JPEG COMPRESSION ALGORITHM USING PROBABILISTIC BOOLEAN LOGIC APPLIED TO CMOS COMPONENTS

DISSERTATION

Christopher I. Allen, Maj, USAF

AFIT-ENG-DS-15-D-001

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-DS-15-D-001

DEMONSTRATION OF INEXACT COMPUTING IMPLEMENTED IN THE JPEG COMPRESSION ALGORITHM USING PROBABILISTIC BOOLEAN LOGIC APPLIED TO CMOS COMPONENTS

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

Christopher I. Allen, B.S.E.E., M.S.E.E.

Maj, USAF

24 December 2015

AFIT-ENG-DS-15-D-001

DEMONSTRATION OF INEXACT COMPUTING IMPLEMENTED IN THE JPEG COMPRESSION ALGORITHM USING PROBABILISTIC BOOLEAN LOGIC APPLIED TO CMOS COMPONENTS

DISSERTATION

Christopher I. Allen, B.S.E.E., M.S.E.E.
Maj, USAF

Committee Membership:

Maj Derrick Langley, PhD
Chairman

Dr. James C. Petrosky
Member

Dr. James C. Lyke
Member

Dr. Mary Y. Lanzerotti
Member

Abstract

The success of Moore's Law has conditioned the semiconductor industry to expect continuing improvements in high performance chips. Limits to the power reduction that can be realized with traditional digital design provide motivation for studying probabilistic computing and inexact methods that offer potential energy savings, performance improvements, and area improvement. This dissertation addresses how much energy and power can be saved if one accepts additional tradeoffs in accuracy (and thus advantages in power consumption, and decreased heat removal).

This work advances the state of the art of inexact computing by optimizing the JPEG File Interchange Format (JFIF) compression algorithm for reduced energy, delay, and area. The dissertation presents a demonstration of inexact computing implemented in the JPEG algorithm applied to an analysis of uncompressed TIFF images of a U.S. Air Force F-16 aircraft provided by the University of Southern California Signal and Image Processing Institute (SIPI) image database. The JPEG algorithm is selected as a motivational example because it is widely available to the U.S. Air Force community and is widely used in many areas including the military, education, business, and users of personal electronics. The JPEG algorithm is also selected because it is by its nature a lossy compression algorithm, where the existence of loss indicates the users willingness to accept error.

The approach of this research is to predict the performance of CMOS components as implemented in solving problems of probabilistic Boolean logic, with a specific focus on the most advanced silicon CMOS technology currently in high volume manufacturing today (the 14 nm FinFET silicon CMOS technology). The main contribution of this research is a method to quantify the energy savings resulting from the decision to accept a specified percentage of error in some components of a computing system. The dissertation presents a demonstration of the JPEG algorithm in which

two components of the algorithm (namely the first and third steps, color space transformation and discrete cosine transform) take advantage of the reduced energy and power that can be achieved when one accepts a certain amount of inexactness in the result. Detailed studies in energy-accuracy tradeoffs in adders and multipliers are presented.

We have shown that we could cut energy demand in half with 16-bit Kogge-Stone adders that deviated from the correct value by an average of 3.0 percent in 14 nm CMOS FinFET technology, assuming a noise amplitude of 3×10^{-12} V^2/Hz. This was achieved by reducing V_{DD} to 0.6 V instead of its maximum value of 0.8 V. The energy-delay product (EDP) was reduced by 38 percent.

Adders that got wrong answers with a larger deviation of about 7.5 percent (using $V_{DD} = 0.5$ V) were up to 3.7 times more energy-efficient, and the EDP was reduced by 45 percent.

Adders that got wrong answers with a larger deviation of about 19 percent (using $V_{DD} = 0.3$ V) were up to 13 times more energy-efficient, and the EDP was reduced by 35 percent.

We used inexact adders and inexact multipliers to perform the color space transform, and found that with a 1 percent probability of error at each logic gate, the letters "F-16", which are 14 pixels tall, and "U.S. AIR FORCE", which are 8 to 10 pixels tall, are readable in the processed image, where the relative RMS error is 5.4 percent.

We used inexact adders and inexact multipliers to perform the discrete cosine transform, and found that with a 1 percent probability of error at each logic gate, the letters "F-16", which are 14 pixels tall, and "U.S. AIR FORCE", which are 8 to 10 pixels tall, are readable in the processed image, where the relative RMS error is 20 percent.

Results presented in this dissertation show that 91.8% of the color space transform can be built from inexact components and that 92.0% of the discrete cosine transformation can be built from inexact components. The results show that, for the case in which the probability of correctness is 99%, the color space transformation has 55% energy reduction per pixel of an uncompressed image, and the discrete cosine transformation step also has a 55% energy reduction per pixel.

Table of Contents

Page

Page

Page

Page

List of Tables

List of Figures

List of Acronyms

AFRL Air Force Research Laboratory

AFIT Air Force Institute of Technology

AWGN Additive White Gaussian Noise

CST Color Space Transformation

DCT Discrete Cosine Transformation

EDP Energy-Delay Product

EOB End of Block

FA Full Adder

FFT Fast Fourier Transform

HA Half Adder

HVS Human Visual System

IC Integrated Circuit

JFIF JPEG File Interchange Format

JPEG Joint Photographic Experts Group

KS Kogge-Stone

KTA Key Technology Area

LET Linear Energy Transfer

LSB Least Significant Bit

MLE Maximum Likelihood Estimation

MSB Most Significant Bit

MSE Mean Square Error

NP Normal Product

PBL Probabilistic Boolean Logic

PCA Potential Capability Area

PDF Probability Density Function

PMF Probability Mass Function

PSD Power Spectral Density

RC Ripple-Carry

RCA Ripple-Carry Adder

RF Radio Frequency

RMS Root-Mean-Square

SEU Single Event Upset

SIPI Signal and Image Processing Institute

SNR Signal to Noise Ratio

SPICE Simulation Program with Integrated Circuit Emphasis

TIFF Tagged Image File Format

USC University of Southern California

WHT Walsh-Hadamard Transform

WPAFB Wright Patterson Air Force Base

List of Symbols

Symbol	Definition
α	compression quality factor
ε	error
$\hat{\varepsilon}$	normalized error
ε_{max}	maximum possible error
A	N-bit input to an adder or multiplier
$\mathcal{A}$	gain of an analog amplifier
a_i	the ith bit of A
B	N-bit input to an adder or multiplier
$\mathcal{B}$	blue component
b_i	the ith bit of B
$\mathfrak{b}$	base of a floating-point number
C	discrete cosine transformation of X
C_{in}	one-bit carry-in input to an adder
C_{out}	one-bit carry-out output of an adder
$\mathcal{C}_b$	chrominance, blue
$\mathcal{C}_r$	chrominance, red
c_i	the ith carry bit within an N-bit adder
$\mathcal{E}$	energy dissipated
$\mathfrak{e}$	exponent of a floating-point number
$\mathfrak{e}_0$	offset bias of the exponent of a floating-point number
$\sim \textbf{Exponential}(\lambda)$	is exponentially distributed with rate parameter λ
f	frequency
$\mathcal{G}$	green component
H	entropy

I_{DD}	power supply current
$\mathfrak{m}$	mantissa of a floating-point number
N	bit width of the inputs to an adder or multiplier
N_{exact}	number of bits which are computed exactly
$N_{inexact}$	number of bits which are computed inexactly
$N_{\mathfrak{e}}$	bit width of floating-point exponent
$N_{\mathfrak{m}}$	bit width of floating-point mantissa
$\sim \mathcal{N}(\mu, \sigma^2)$	is normally distributed with mean μ and variance σ^2
P	product of a multiplier
$\tilde{P}$	approximate product
$\breve{P}_k$	kth partial product of a multiplier
$\mathrm{P}(\cdot)$	probability
$\mathcal{P}$	power dissipated
$\mathcal{P}_d$	dynamic power
$\mathcal{P}_s$	static power
p	probability of correctness
$\breve{p}_{k,i}$	ith bit of $\breve{P}_k$
Q	quantized form of C
$\breve{Q}$	zigzag arrangement of Q
$q_{i,j}$	(i,j)th element of Q
R	compression ratio
$\mathcal{R}$	red component
S	N-bit sum of an adder, excluding the carry-out bit
S^+	$(N+1)$-bit augmented sum of an adder, including the carry-out bit
T	clock period
t	time

U	unitary discrete cosine transformation matrix
$\sim \mathbf{Unif}(a, b)$	is uniformly distributed between a and b
V_{DD}	power supply voltage
w	vector of all noise sources within an inexact circuit
w_{in}	noise at the input node of a circuit
w_{out}	noise at the output node of a circuit
X	8×8 tile containing image (Y, C_b, or C_r) data
X	input vector to a digital logic circuit
Y	output of a digital logic circuit
$\mathcal{Y}$	luminance
Z	quantization factor matrix
$z_{i,j}$	(i, j)th element of Z

Subscripts

d	dynamic (as in dynamic power)
i	ith bit position
in	input
k	kth stage of a circuit
l	lth state in a sequence
max	maximum
min	minimum
out	output
rms	root-mean-square average
s	static (as in static power)

Superscripts

$+$	augmented (as in the augmented sum of an adder)

Accents

$\breve{}$	partial (as in partial product)
$\check{}$	zigzag sequence
$\hat{}$	normalized
$\bar{}$	average (mean)
$\tilde{}$	pertaining to an approximate or inexact computational circuit

DEMONSTRATION OF INEXACT COMPUTING
IMPLEMENTED IN THE JPEG COMPRESSION ALGORITHM
USING PROBABILISTIC BOOLEAN LOGIC
APPLIED TO CMOS COMPONENTS

I. Introduction

The success of Moore's law [1, 2, 3] has conditioned the semiconductor industry to expect continuing improvements in high performance chips. The success has produced a situation in which circuit designers are designing products around Moore's Law. Concerns about the possible end of Moore's Law are being raised, and researchers are seeking ways to sustain this trend. This trend encroaches on areas such as probabilistic computing [4, 5, 6, 7, 8, 9] or neuromorphic computing [10, 11, 12, 13]. The general motivation for studying probabilistic computing and inexact methods lies in three areas of potential energy savings, performance improvement, and area improvement (that is, reduced density).

While Moore's Law has produced an expectation that more transistors can be packed on a chip, there is a physical limit to classical scaling theory described by Dennard [14]. At the same time it is difficult to reliably manufacture billions of transistors that operate without failure on a single chip.

With an insatiable demand for computation, there are limits to the power reduction that can be realized with traditional digital design. There is also a need to build a fault tolerant chip that can "handle failure gracefully" [15]. Because of these limits, researchers are investigating methods such as probabilistic computing as ways to extract functionality that is "good enough" while operating with less power.

There are two broad approaches to studying inexact methods. First, there are methods that provide deterministic inexactness (that is, a truth table that is occasionally wrong). It is possible to apply these methods in situations when a system—in which the logic is contained—does not care about a wrong answer [16]. That is, deterministic inexactness can be useful when the system doesn't care about the wrong result.

Second, there is the method of nondeterministic inexactness. Nondeterministic inexactness refers to a circuit design that has a certain probability of error due to unknown variables including noise, interference, manufacturing defects, or radiation. Note that not all error is created equal. There is error due to approximation, and there is random error. There are profoundly different consequences of these types of error. For example, some error may be tolerable when processing image data or audio data.

Noise is one of the sources of inexactness, such as when noise corrupts the truth table. Circuit designers intend to design chips to be very robust. However, when circuit designers start to push the boundary, and they start by degrading the design itself, they are taking these steps because they may be able to eliminate some transistors or (perhaps) use smaller transistors. In such situations, circuit designers start bending some of the rules of good design practice, because they are trying to achieve a result that is "good enough".

Both deterministic and nondeterministic inexactness must be well understood by the circuit designer. Specifically, the designer needs to understand when and if inexactness can be applied. Consider the situation in which a designer has prepared a block diagram of a system. There are circumstances in which designers can inspect a block diagram and decide which blocks of the design are those in which they can use inexactness and which blocks are those in which they cannot use inexactness.

For example, it is undesirable for a state machine to jump to a random next state. Therefore circuit designers would assume that that part of the design (that is, the state machine) should be handled by traditional design methods—that is, by exact methods. But the designers may identify other parts of the system design in which one may realistically use inexactness. Here, the fundamental point of this argument is that circuit designers cannot use inexactness indiscriminately, and inexactness only works in certain types of circumstances.

The main contribution of this dissertation is a method to quantify the energy savings resulting from the decision to accept a specified percentage of error in some components of a computing system. Recall von Neumann's pursuit to build reliable systems from unreliable components [17, 18, 19, 20]. Von Neumann struggled with the idea of how to obtain correct results when the constituent components are unreliable. This work was performed following the invention of the transistor, and vacuum tubes were used but failed (with vacuum tubes, the mean time to failure was 10 to 20 minutes, and computing systems contained many vacuum tubes). Designers started to worry less when transistors provided higher yield, but out of the work by von Neumann emerged ideas such as triple module redundancy.

In computing applications, data compression is necessary due to memory, disk space, and network bandwidth constraints. The Joint Photographic Experts Group (JPEG) File Interchange Format (JFIF) is ubiquitous, and is a very effective method of image compression. Throughout this document, we refer to the JFIF compression algorithm, which is used for this research, as the "JPEG compression algorithm" (not to be confused with JPEG2000). For many imaging applications, energy is at a premium due to battery life and heat dissipation concerns. For space applications, the electronic systems are also susceptible to the effects of the natural radiation environment; for example, solor protons and galactic cosmic rays can cause single

event upsets within the circuits, which is another form of hardware unreliability. The emerging field of inexact computing promises to deliver energy-parsimonious computation by sacrificing the strict requirement for accuracy in digital logic circuits. The contribution of this research will be to advance the state of the art of inexact computing by optimizing the JPEG compression algorithm for reduced energy, delay, and area.

Inexact computing is ultimately based on information theory. Consider an adder which computes a sum S^+ from inputs A and B. Now consider an approximate adder with output $\tilde{S}^+$, which is an estimator for S^+. How much information does $\tilde{S}^+$ contain about about the desired sum S^+? With inexact computing, it is possible to save energy, delay, and area, and still obtain information about S^+, without obtaining the exact value of S^+.

1.1 Strategy for Applying Inexact Methods

How do we know whether an inexact design is appropriate for a particular application? The decision-making process is outlined in the flowchart in Fig. 1. The flowchart proceeds as follows. First, the designer must consider the algorithm which the system will execute. If the algorithm is well-understood an has been previously implemented in hardware using exact methods, then historical data will provide a baseline for the performance, area, and energy consumption required to implement the algorithm. If the algorithm is new, and has never previously been built into hardware, then the designer should create an exact design (i.e. schematics or layouts) to determine the baseline for the algorithm.

From the baseline, the designer can estimate the whole algorithm will occupy an estimated fraction f of the chip. The fraction f could be a fraction of the total chip area, gate count, or other such metric. In the "profiling" step, the designer divides

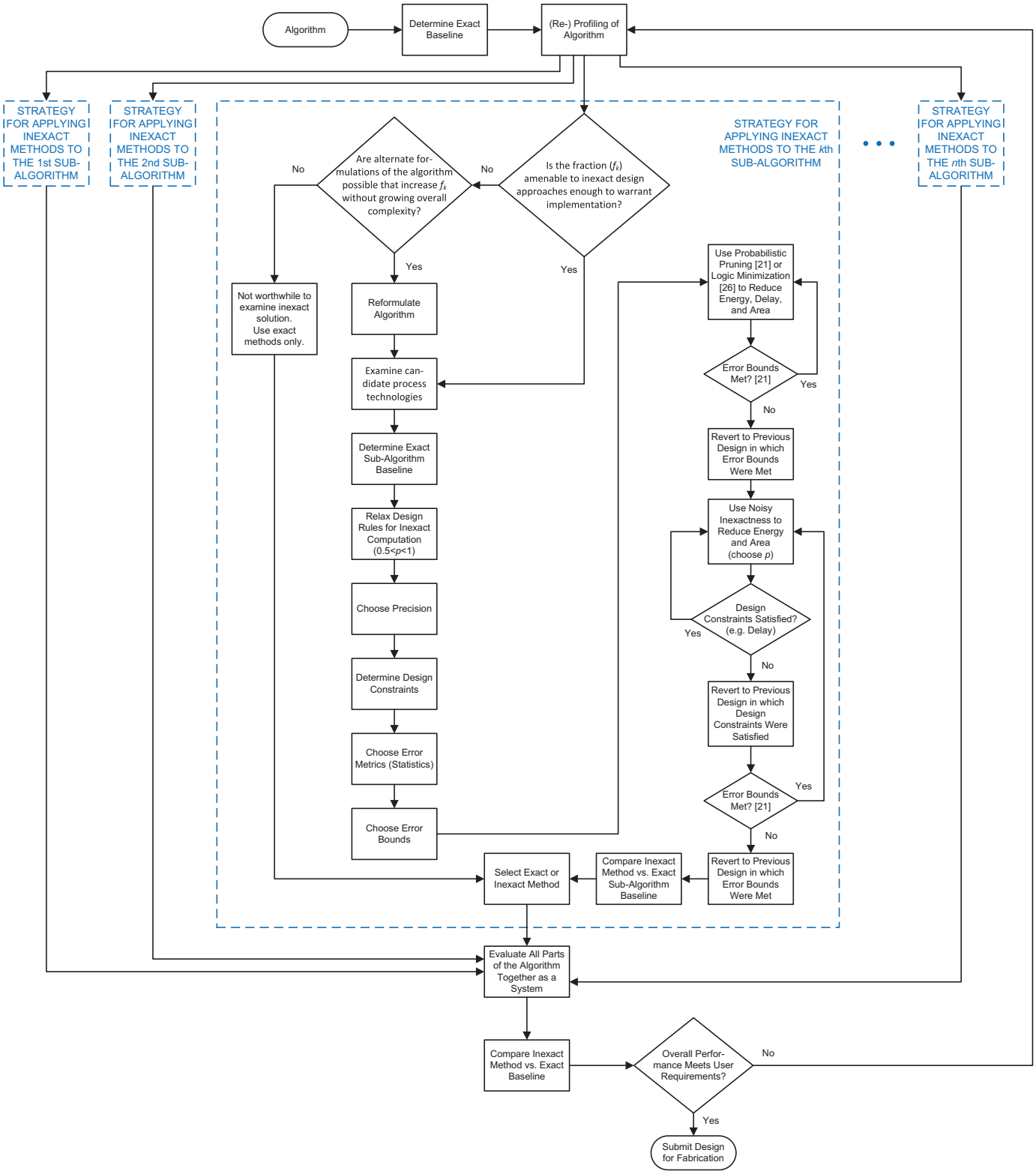


Figure 1. Decision-making flowchart for inexact design.

the algorithm into sub-algorithms. If there are n sub-algorithms, then the kth sub-algorithm occupies an estimated fraction f_k of the chip. The total of all fractions f_k add up to f, that is,

$$\sum_{k=1}^{n} f_k = f. \tag{1}$$

Each sub-algorithm is designed separately. Types of sub-algorithms include: computation, state machines, encoding, routing, memory storage and retrieval, time synchronization, and random number generation. Some sub-algorithms are potentially amenable to inexactness, while others are not. Of these categories, we would argue that computation, memory storage and retrieval, and random number generation could possibly be done inexactly in a useful way, while the others would not. However, future research may find useful ways to perform the other sub-algorithms inexactly also.

Some sub-algorithms have multiple possible formulations. If an inexact design is being proposed, then the designer should choose the formulation that is most amenable to inexact methods; for example, a formulation based on addition is likely more tolerant of inexactness than a formulation based on a decision tree. If, after considering all possible formulations of the sub-algorithm, there appears to be no possible benefit to an inexact design, then the designer will choose to design the sub-algorithm using exact methods only.

If the algorithm is a potentially inexact algorithm, then it can be implemented in an inexact hardware technology. As device dimensions shrink, it is increasingly difficult for devices to perform consistently, reliably, and to have good noise margins. Future hardware technologies may be designed to perform in a probabilistic manner, that is, it will function "correctly" most of the time, but at each individual node there is a probability p of correctness, where $0.5 < p < 1$, and a probability $(1 - p)$ of error. Errors could be caused by noise, radio frequency (RF) interference, crosstalk between

components, threshold voltage variations, cross-chip variations, or cosmic radiation. In inexact hardware technology, the interconnect between transistors would also be susceptible to crosstalk, which is a possible error source. Such technology would include relaxed design rules with tabulated probabilities of correctness which depend on the area and spacing allocated to each transistor or interconnect.

Based on the exact algorithm baseline and the candidate process technologies, the designer should determine a baseline for the performance of the sub-algorithm implemented using exact methods. This is used as a basis for comparison with the inexact design.

The next step is to choose the number of bits of precision required to process the data. Information collected from analog sensors is inherently uncertain, and needs only a limited number of significant figures to express it. This should be reflected in the computing hardware. For example, if the significant figures of the data can be expressed in eight bits, then using a 64-bit adder to process it would be a waste of hardware and energy.

The designer must also determine the constraints which the design must meet. Constraints may include timing, area, current, and power dissipation.

Next, the designer chooses the metrics by which the system will be evaluated. These may include the energy consumption, delay, and area of the hardware. Other metrics will reflect the overall quality of the overall system output. For example, in image compression, the metrics for system output include root-mean-squared (RMS) error, Signal to Noise Ratio (SNR), and compression ratio. For human consumption of images, however, simple metrics do not adequately summarize the quality of an image, so it is up to the users opinion as to whether the image quality is "good enough".

The next step is to choose the error tolerance, or error bounds, for the overall

system output. Error metrics could include statistics such as: the maximum possible error, RMS error, or likelihood of nonzero error. The system design will not be allowed to exceed the error tolerance.

Probabilistic pruning and probabilistic logic minimization techniques are methods of simplifying the digital logic hardware while producing an approximate computational output in a deterministic manner. Probabilistic pruning and probabilistic logic minimization simultaneously meet the objectives of reducing energy consumption, delay, and chip area, by sacrificing the accuracy of the output in some cases. These techniques are used repeatedly until a maximum error tolerance is met [21].

After using deterministic inexact design techniques, the designer considers the effects of non-deterministic errors due to the inexact nature of the process technology. The designer can "tune the design to reduce area and energy consumption, until the effects of non-deterministic errors exceed the pre-determined error bound, or the design constraints (e.g. delay) are no longer met.

After creating the inexact design for the sub-algorithm, the designer compares the inexact design with the baseline previously determined. If the inexact design results in a substantial savings of energy, delay, area, energy-delay product, or other such metric, without an excessive amount of error, then the designer chooses to use the inexact design for the sub-algorithm. If, however, there is little apparent benefit to the inexact design, then the designer chooses to use an exact design instead.

Once hardware has been designed for all sub-algorithms, the parts must be evaluated together as a whole system. At this point, the designer makes sure the entire algorithm works properly, and then compares it to the exact baseline for the whole algorithm. If the overall performance meets user requirements, then the design is submitted for fabrication. If not, then the designer must re-profile some or all of the algorithm, and then re-design some or all of the sub-algorithms until performance is

satisfactory.

1.2 Motivational Link to Air Force Needs and Vision

The need in the U.S. Air Force for the capability to create perfect computing systems out of imperfect components to achieve military and space objectives dates at least to the construction of the earliest computing systems including those built from mechanical components, vacuum tubes, and instrumentation in the earliest Apollo missions. There are some applications of electronic systems that can tolerate inexact systems, and some applications that cannot tolerate inexactness.

For example, for a life support system, the system should be as reliable as possible, and inexactness is undesirable. For a satellite system that can operate in low power mode, inexact computing may be tolerated to achieve trade-offs in size, weight, and power. To incorporate inexact systems in space applications, there is a need to quantify trade-offs in size, weight, and power.

This dissertation is concerned with exploring the incorporation of inexactness in the JPEG algorithm, shown in Fig. 2, which is by its nature a lossy compression algorithm, where the existence of loss indicates the user's willingness to accept error. That will be discussed in this dissertation. As a contribution of this work, we find that the Color Space Transformation (CST) has 55% energy reduction per pixel of an uncompressed image, and the discrete cosine transformation step also has a 55% energy reduction per pixel for the case in which the probability of correctness $p = 0.99$. These results are promising and indicate that JPEG can be considered a fitting example for this type of study. Future work will consider the additional components, which are: tiling, quantization, zigzagging, run-amplitude encoding, and Huffman encoding. The JPEG algorithm can be implemented in hardware or software. Tiling and zigzagging are simply routing of data, and in a hardware implementation, it is

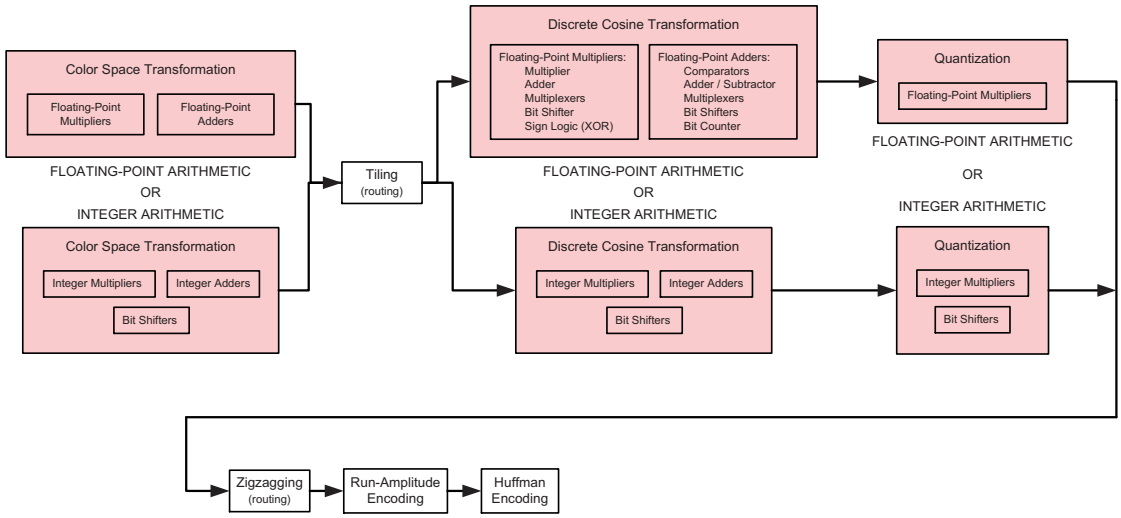


Figure 2. Block diagram of the JPEG image compression algorithm. In this dissertation, the JPEG algorithm is a motivational example for inexact computing. The shaded boxes show areas where inexact methods can be considered (for example, in adder circuits and multiplier circuits). The white boxes show areas where inexact methods cannot be considered ("keep-out zones").

possible to implement these two steps via wiring only. In a software implementation, we would not want errors in the instruction pipeline or memory addressing, so in that case inexact tiling and zigzagging would not be desirable. In the JPEG algorithm, the user's willingness to accept error leads to trade-offs in size, weight, and power within the color space transformation, discrete cosine transformation, and quantization steps, as indicated in the pink areas in Fig. 2.

We now consider briefly a historical example of why the Air Force should care about implementing inexactness in computing systems. In the book entitled *Digital Apollo*, the author David Mindell, Dibner Professor of the History of Engineering and Manufacturing, Professor of Engineering Systems, and Director of the Program in Science, Technology, and Society at MIT, describes how the Apollo astronauts became their own "ultimate redundant components" by carrying on board with them extra copies of the computer hardware so that in case of failure, they could replace the faulty hardware with one of the backups. Triple module redundancy is one example of a now-

common approach to creating more perfect computing systems out of components that are understood to be imperfect.

In the book, he writes,

> "Robert Chilton remembered that Hall and his group paid constant attention to reliability questions, though NASA wasn't prepared to give them a specification....[more specific than] 'as reliable as a parachute' ... with one in a hundred chance the mission would fail, and a one in a thousand chance the astronauts would not survive." (page 128, Mindell)
>
> "Hall estimated that his Block I integrated circuit computer would have a reliability of 0.966 ...., but the spec he had been given required reliability nearly ten times better. To make up the difference he proposed [relying] on the skills of the astronauts: they would repair the computer during flight.... Hall proposed that Apollo flights also carry a special machine, a 'MicroMonitor,' a smaller version of the equipment used to check out the computer on the ground. The device was heavy and took up space, required its operator to 'exercise considerable thought,' and required the operator to have a mere three to six months of training... 'This device is known to be effective, Hall wrote, 'in direct proportion to the training and native skill of the operator'." (page 129, Mindell)
>
> "Astronauts had been billed as the ultimate redundant components. Asking them to improve the reliability of their equipment seemed sensible, but it proved no simple task." (page 130, Mindell) [22]

In December 1965, Eldon C. Hall, of the MIT Instrumentation Laboratory in his report E-1880 entitled, "A Case History of the AGC Integrated Logic Circuits" [23], Mr. Hall explains in this report that the use of "one single, simple integrated circuit for all logic functions" was required to achieve the goals of "low weight, volume, and power coupled with extreme high reliability." As he describes, the "one single, simple integrated circuit" was the three input NOR gate (the NOR3 logic gate). He writes about the following tradeoffs in the selection of the logic element in the Apollo Guidance Computer,

> "The logic element utilized in the Apollo Guidance Computer is the three input NOR Gate... At the time that the decision was made to use in-

> tegrated circuits, the NOR Gate was the only device available in large quantities. The simplicity of the circuit allowed several manufacturers to produce interchangeable devices so that reasonable competition was assured. Because of recent process development in integrated circuits, the NOR Gate has been able to remain competitive on the basis of speed, power, and noise immunity. This circuit is used at 3V and 15mW, but is rated at 8 V and 100 mW. Unpowered temperature rating is 150 degrees C. The basic simplicity of the three input gate aids an effective screening process. All transistors and resistors can be tested to insure product uniformity. The simplicity of the circuit also aids in the quick detection and diagnosing of insidious failures without extensive probing as required with more complicated circuits." (page 4, Hall 1965).

It is recognized from at least the time in this report (1965) that the inherent reliability gains must be implemented in the design stages of the computer. This dissertation recognizes that tradeoffs between energy, power, and reliability continue even with the most advanced silicon CMOS technologies available today. One can even state that the tradeoffs between energy, power, and reliability are most especially of concern to the U.S. Force today, because the needs of the silicon CMOS technologies are predominantly, and increasingly, driven by the consumer marketplace much more so than the environment in which the Apollo missions found themselves when the DoD formed a greater portion of the demand for integrated circuits.

Today, however, the "strategic environment that the Air Force faces over the next two decades is substantially different from that which has dominated throughout most of its history," as explained in the document entitled, 'A Vision for Air Force Science Technology During 2010-2030" [24]. This document provides an overview of the role of 110 Key Technology Areas (KTAs) including the KTA of advanced computing architectures that support the 12 Potential Capability Areas (PCAs) of the U.S. Air Force.

The document "A Vision for Air Force Science Technology During 2010-2030" also describes advantages and additional capabilities that reductions in power, performance, and area can provide in electronics in order to enable the U.S. Air Force to

achieve superiority in air, space, and cyberspace domains. Specific examples are persistent near-space communications relays (page 63), where power and thermal management (heat dissipation) challenges exist. Reductions in size, weight, power, and thermal management requirements will further enable greater integration of complex systems in advanced fighters, space satellites, and "other tactical platforms." (page 72).

This dissertation presents a demonstration of inexact computing implemented in the JPEG compression algorithm using probabilistic Boolean logic applied to CMOS components, with a specific focus on the most advanced silicon CMOS technology currently in high volume manufacturing today (namely, the 14nm FinFET silicon CMOS technology). The JPEG algorithm is selected as a motivational example since it is widely accessible to the U.S. Air Force community. It is also well known that the JPEG algorithm is widely known and widely used worldwide, in many areas including but not limited to the military, education, business, and by people of ages who capture images on their personal electronics, cameras, and cell phones. This dissertation is interested in the question about how much energy and power can be saved if one might be able to accept additional tradeoffs in accuracy (and thus lead to potential advantages such as lower power consumption, increased battery life, and decreased need to dissipate heat since the power consumption is reduced).

The goal of this dissertation is to present a demonstration of this JPEG algorithm in which two components of the algorithm (namely the first step, color space transformation, and the third step, discrete cosine transformation) take advantage of the reduced energy and power that can be achieved when one accepts a certain amount of inexactness in the result. Energy-accuracy tradeoffs in adders and multipliers are explored in detail, and detailed results are presented quantifying the extent to which the power-delay product can be reduced as a function of probability of correctness.

The dissertation applies the inexact JPEG algorithm to an analysis of uncompressed TIFF images of an F-16 U.S. Air Force plane provided by the University of Southern California, as shown in Fig. 3. In this dissertation, we only analyze the data from the intensity (Y) component of the image, so in Chapter V the figures appear in black-and-white; however, we expect very similar results for the color components (Cb and Cr) since they are processed in a very similar way. The results quantify tradeoffs between the probability of correctness, the SNR, and the Root-Mean-Square (RMS) error. Specifically the results show that as the probability of correctness takes on a smaller value (decreases), the SNR takes on a smaller value, and the RMS error increases. Values are quantified for each of the tradeoffs.

1.3 Contributions

We have shown that we could cut energy demand in half with 16-bit Kogge-Stone adders that deviated from the correct value by an average of 3.0 percent in 14 nm CMOS FinFET technology, assuming a noise amplitude of 3×10^{-12} V^2/Hz (see Fig. 32). This was achieved by reducing V_{DD} to 0.6 V instead of its maximum value of 0.8 V. The energy-delay product (EDP) was reduced by 38 percent (see Fig. 33).

Adders that got wrong answers with a larger deviation of about 7.5 percent (using $V_{DD} = 0.5$ V) were up to 3.7 times more energy-efficient, and the EDP was reduced by 45 percent.

Adders that got wrong answers with a larger deviation of about 19 percent (using $V_{DD} = 0.3$ V) were up to 13 times more energy-efficient, and the EDP was reduced by 35 percent.

We used inexact adders and inexact multipliers to perform the color space transform, and found that with a 1 percent probability of error at each logic gate, the letters “F-16”, which are 14 pixels tall, and “U.S. AIR FORCE”, which are 8 to 10

Figure 3. Original uncompressed image of an F-16, file name 4.2.05.tiff, from the USC SIPI image database [25].

pixels tall, are readable in the processed image, as shown in Fig. 40f, where the relative RMS error is 5.4 percent.

We used inexact adders and inexact multipliers to perform the discrete cosine transform, and found that with a 1 percent probability of error at each logic gate, the letters "F-16", which are 14 pixels tall, and "U.S. AIR FORCE", which are 8 to 10 pixels tall, are readable in the processed image, as shown in Fig. 41f, where the relative RMS error is 20 percent.

In the next section, we present a literature review of inexact computing and describe prior work. This section provides background information regarding the approach for inexact computing that is used in this dissertation.

II. Literature Review

Inexactness has typically been understood to be inherent in analog circuits, but not in the conventional understanding of digital logic. Previous work in the field of inexact digital CMOS has focused on two types of inexactness: (1) circuits which produce an approximate result, but are deterministically erroneous by design, and (2) circuits which suffer from the effects of random noise. The first type of inexactness is achieved via probabilistic pruning [21] or probabilistic logic minimization [26]. Probabilistic pruning is a bottom-up approach in which components are removed from the schematic of a circuit, for the purpose of saving energy, delay, and area, while producing output which is "correct" in the majority of cases. Probabilistic logic minimization accomplishes the same objective of saving energy, delay, and area by creating an erroneous, but simpler, design based on a modified truth table which is "correct" in the majority of cases. Probabilistic pruning and probabilistic logic minimization both produce an approximate result, constrained by a desired error bound [21, 26]. The designer chooses the error bound that meets the needs of the system. Probabilistic pruning and probabilistic logic minimization enable the designer to reduce energy consumption, delay, and chip area by creating a circuit which is simpler than the conventional (exact) circuit.

From the perspective of this dissertation, inexact computing is contrary to the notion of error detection and correction. Whereas the primary goal of inexact computing is to reduce energy consumption [27], error detection and correction techniques contain additional components which increase the energy, delay, and area of the circuit. These techniques could be used in conjunction with inexact computing, but only if the overall energy savings outweigh the additional costs. For example, triple module redundancy could be used if the energy consumption of the inexact circuit is less than $1/3$ the energy consumption of the equivalent exact circuit.

The second type of inexactness, which is non-deterministic (noise-susceptible) inexactness, has been achieved by severely lowering the power supply voltage, thus reducing the noise margins of the circuit [28, 29, 30]. Great energy savings can be achieved this way, if the designer is willing to tolerate the error. For example, for an inverter in 0.25 μm CMOS technology with an RMS noise magnitude of 0.4 V, [27] reports a 300% energy reduction per switching cycle can be achieved by allowing the probability of correctness p to drop from 0.99 to 0.95.

Prior work investigated energy and performance with the use of high level C-based simulations [31, 32, 33]. These papers treat individual logic gates as unreliable, with an associated probability of correctness $p < 1$, or equivalently a finite error probability $1 - p$, and present simulation results of complex circuits built out of unreliable primitive elements. Additional prior work used circuit simulations of a 32-bit weighted voltage-scaled adder with carry look-ahead capability and demonstrated a calculation error of 10^{-6} while reducing the total power consumption by more than 40% in 45 nm CMOS FDSOI technology [34]. This shows a promising approach to the study of inexact adders, which we have used in our work. Additional prior work shows a four-fold reduction in energy-delay product using probabilistic pruning of 64-bit Kogge-Stone adders, at the expense of an 8% average error [35]. In this dissertation, we use similar metrics to evaluate adders.

In some cases, erroneous bits inside a circuit have no impact on its final output. Researchers are interested in "don't care sets" which describe sets of erroneous inputs that don't cause errors at the output ("observability don't care" conditions), or input vectors that can never occur ("satisfiability don't care" conditions) [16, 36]. This dissertation does not take that approach; however, it is clearly relevant to the field of inexact computing. For example, if a designer is aware of the observability don't care conditions of a device, then he could simplify the circuit using probabilistic

logic minimization without causing any errors at the output. As another example, random noise inside a digital circuit could cause errors internal to the circuit without affecting the output; based on the observability don't care conditions, the designer could choose to use unreliable components in those areas of the circuit in which errors would be unlikely to affect the output. On the other hand, while satisfiability don't care conditions may exist for an error-free logic circuit, they may not exist if that circuit is susceptible to random errors, i.e. random noise may cause unexpected input conditions.

The approach of this literature review is informed by the goal of this dissertation. The approach of this work is to investigate the energy reduction and energy-delay product in a selection of adder and multiplier architectures made using unreliable logic gates, and then to build an inexact JPEG compression algorithm using these inexact adders and multipliers. In the interests of high-speed simulation and of collecting large sample sizes, these simulations were performed with Matlab using a Probabilistic Boolean Logic (PBL) error model.

The PBL error model is more simplistic than an analog error model. In this dissertation, we also compute the energy and Energy-Delay Product (EDP) of selected adder architectures in 14 nm FinFET CMOS technology as a function of error. This is similar to the approach used in [34]. As an example of the approach presented in this dissertation, consider a circuit design that exhibits a 20% error rate. The results in this dissertation show that for the specific circuits considered with an error rate is 20%, the payback for accepting a 20% error rate is an energy reduction of 90%. If a circuit designer were willing to tolerate 20% error and gain this energy, one can then ask the question, 'what good is this'? For example, one could implement triple module redundancy in either temporally or spatially: temporally, one could sample the same data at three different points in time; spatially, one could replicate the

circuit design three times, with three copies, knowing that statistically the majority vote will always be correct. The main point is that when one can improve the accuracy through the use of redundancy and, at the same time, save so much energy that one obtains practically a perfect answer, then accepting the error is worth it.

III. Background

In this section, we present a taxonomy of inexact computing. We review integer adder architectures and show how the model of PBL can be applied to adders. We explain how PBL can be used within a binary or an analog circuit model, and introduce nomenclature used throughout the dissertation. Next, we review integer multiplier architectures. We review probability distributions and Maximum Likelihood Estimation (MLE), and explain how they can be used to characterize the error distributions of inexact adders and multipliers. Finally, we provide a detailed review of the JPEG compression algorithm.

The following definitions are useful for understanding the framework of inexact computing:

- Inexact: Probabilistic methodology for determining an answer.
- Imprecision: Small uncertainties about the LSBs of the answer. The maximum possible error that can occur. Imprecision is inherent in data collected from analog sensors.
- Probability of Correctness: Determines the dispersion (i.e. standard deviation) of the errors.
- Erroneous: Failure of the system to compute a useful answer.

3.1 Taxonomy of Inexact Computing

3.1.1 Deterministic.

Many different sources of deterministic error are possible. Error can be measured in many different ways. The amount of error to tolerate is a design parameter for the inexact system. Error sources include:

- Limited precision of the processor (number of bits)
- Inexact design techniques, such as probabilistic pruning or probabilistic logic minimization

3.1.1.1 Limited Precision.

Digital computers are inherently inexact, because they have limited precision. For example, using double-precision (64-bit) floating-point numbers, $\frac{1}{6}$ is approximated as 0.166666666666666660. If we add that approximation to itself six times, the result is slightly less than 1. For practical purposes, data being analyzed do not require 64 bits of precision. Furthermore, analog data are inherently bounded by a range of uncertainty, and this uncertainty carries into the digital domain when analog data are digitized. Data collected from actual experiments do not have infinite precision; the numbers have a limited number of "significant figures" or *bits of information.* Whereas a computer may store a piece of experimental data in a 64-bit register, only the first eight bits (for example) may contain meaningful information based on the experiment.

The limits of the information content carry forward from the source data into other data computed from it. If we have an N_A-bit number A which contains $N_{I(A)}$ bits of information, and an N_B-bit number B which contains to $N_{I(B)}$ bits of information, then the product AB has $N_A + N_B$ bits, but only $N_{I(P)}$ bits of information, where $N_{I(P)}$ is the lesser of $N_{I(A)}$ and $N_{I(B)}$. The information content of the sum $A + B$ is limited by the greater of $LSB_{I(A)}$ and $LSB_{I(B)}$, where $LSB_{I(A)}$ is the least significant bit of A containing meaningful information, and $LSB_{I(B)}$ is the least significant bit of B containing meaningful information.

The point of this is that we do not need to build a 64-bit adder if an 8-bit adder can handle the information content. Obviously, adders and multipliers of 64-bit numbers

occupy more area on chip, have longer delay time, and consume more energy than adders and multipliers of 8-bit numbers (all else being equal). Since the objectives of inexact computing include savings of energy, delay, and area, eliminating unneeded precision is consistent with the philosophy of inexact computing. Details about this technique applied to the JPEG compression algorithm are explained in Section 4.6.

3.1.1.2 Probabilistic Pruning.

Probabilistic pruning is a bottom-up, architecture-level approach to inexact computing [21, 35]. It is accomplished by taking an exact design, and then deleting those logic gates which are least used and have the least impact on the overall accuracy of the system. Deleting components from the design, like pruning leaves off a tree, reduces the delay, area, and power consumption of the system. The result is a computer which is inaccurate by design, but only in a limited number of cases which occur infrequently. To decide which components to prune, the designer looks at each element in the circuit and considers: (1) the probability of the element being active at any given time, and (2) the magnitude of the error that would result from deleting that circuit element. After pruning a component, the designer then "heals" the floating inputs of the remaining elements that were previously connected to the pruned element, as described in Section 3.1.1.2.3. The designer continues pruning the circuit until the error of the pruned circuit exceeds a desired error bound ($\bar{\varepsilon}_{max}$).

3.1.1.2.1 Probability of an Element Being Active.

The probability of a logic element being active depends on the application, and is determined by analysis, modeling, or simulation. Specifically, the designer must predict the probability of every possible input vector to the circuit. This depends on the expected data set. Then the designer can determine the accuracy penalty which would result from pruning a circuit element.

3.1.1.2.2 Quantifying Error.

The designer chooses an error metric depending on the application. Lingamneni [21] defines three possible metrics: average error, error rate, and relative error magnitude. Additionally, these error metrics can be weighted or uniform (unweighted). In the unweighted case, all bits have equal weight when calculating error. In the weighted case, the jth bit of a binary number is assigned a weight factor η_j equal to 2^j.

Average error: The average error of a pruned circuit $\mathcal{G}'$ relative to an exact circuit $\mathcal{G}$ is computed as [21]:

$$\mathrm{Er}(\mathcal{G}') = \sum_{k=1}^{\mathcal{V}} p_k \times |\tilde{Y}_k - Y_k| \leq \bar{\varepsilon}_{max} \tag{2}$$

where:

$\mathcal{V}$ = the number of possible input vectors; or else the number sampled
$< Y_{k,1}, Y_{k,2}, \ldots Y_{k,n} >$ = output vector of exact circuit
$< \tilde{Y}_{k,1}, \tilde{Y}_{k,2}, \ldots \tilde{Y}_{k,n} >$ = output vector of pruned circuit
n = number of bits
p_k = probability of occurrence for each input vector
$\bar{\varepsilon}_{max}$ = desired error bound
$\eta_j = 2^j$ = weight factor of the jth bit of Y_k and $\tilde{Y}_k$,
or $\eta_j = 1$ for all j (unweighted error model)

Error rate:

$$\begin{aligned} \text{Error Rate} &= \frac{\text{Number of Erroneous Computations}}{\text{Total Number of Computations}} \\ &= \frac{\mathcal{V}'}{\mathcal{V}} \end{aligned} \tag{3}$$

Relative error magnitude:

$$\text{Relative Error Magnitude} = \frac{1}{\mathcal{V}} \sum_{k=1}^{\mathcal{V}} \frac{|Y_k - \tilde{Y}_k|}{Y_k} \tag{4}$$

By pruning away circuit elements which are seldom active, or which have little effect on the final output, the errors from Equations (3)-(4) will be small. To obtain maximum savings in energy, delay, and area, the designer will continue pruning until the average error reaches the desired error bound ($\bar{\varepsilon}_{max}$). To predict error rates for complex circuits it may not be practical to compute the errors across the entire input space; therefore, a random sample may be chosen.

3.1.1.2.3 Healing.

After pruning away a circuit element, the inputs of some of the remaining circuit elements will be floating and undefined. The designer can heal each floating input in one of three ways: (1) connect them to ground, (2) connect them to the supply voltage V_{DD}, or (3) connect them to one of the inputs of the pruned element. The best choice is whichever one minimizes the error.

3.1.1.3 Probabilistic Logic Minimization.

Probabilistic logic minimization is a top-down, architecture level approach to inexact design [26]. In this method, the designer looks at the truth table of an exact circuit, and then considers flipping bits in ways that make the logic simpler. This results in a circuit which is occasionally erroneous, but has less delay, area, and power consumption than the exact circuit. As in the probabilistic pruning method, the goal is for the errors to be infrequent and small in magnitude. There is no hardware overhead to this technique.

As an example, consider the carry-out function of a one-bit full adder. The exact

logic is

$$c_{out} = ab + bc + ac. \tag{5}$$

The truth table for this function has eight possible outputs. Equation (5) can be approximated by

$$c_{out} = ab + c \tag{6}$$

or

$$c_{out} = a(b + c) \tag{7}$$

or

$$c_{out} = ab + bc + ac + \overline{abc}. \tag{8}$$

in each case, the truth table is incorrect in one of the eight positions. However, while Equations (6)-(7) are simpler than (5), Equation (8) is more complicated. Therefore, (6)-(7) are favorable bit flips and are good candidates for probabilistic logic minimization, while (8) is unfavorable and would not be used. The errors due to probabilistic logic minimization can be quantified using Equations (2-4).

3.1.1.4 Don't Care Sets.

Probabilistic pruning and probabilistic logic minimization each create approximate solutions to digital logic problems. For some input vectors there will be zero error at the output, and in other cases there will be some error. There is considerable research [16] regarding the identification of "don't care" sets of input or output vectors:

- A satisfiability don't care condition is an input vector that can never occur. For example, if a digital circuit performs a function $f(a, b)$ on a binary inputs a and b, and $a = b$ OR c, then $(a, b) = (0, 1)$ is an impossible input vector to f.

- An observability don't care condition occurs when changes to the input vector

do not change the output vector. For a digital vector function f of input vector X, observability don't care means $\partial f(X)/\partial X = 0$.

Don't care analysis could be used to identify all the zero-error conditions resulting from probabilistic pruning or probabilistic logic minimization. However, inexact computing extends this idea by allowing a nonzero error distribution to be introduced into the digital system.

3.1.1.5 Mutual Information: The Usefulness of the Estimator.

A skeptic will ask: Given that an inexact signal is *wrong*, how is it any better than random noise? Or why not just output zeros all the time? The answer is that the inexact signal contains more *information* than a random signal or a zero. Using a concept called *mutual information*, we can quantify the usefulness of an inexact signal relative to an exact signal.

From information theory, we define the entropy H as the average uncertainty of a random variable [37]. For a discrete random variable X, entropy can be expressed as the number of binary digits required to quantify any possible outcome of X. Entropy is calculated as

$$H(X) = -\sum_{\text{all } x} p(x) \log_2 p(x), \tag{9}$$

where $p(x)$ is the probability mass function of the discrete random variable X. If X is uniformly distributed over 2^N possible values, then $H(X) = N$ bits, and within the framework of information theory, N is not necessarily an integer.

The joint entropy $H(X, Y)$ of two discrete random variables X and Y is

$$H(X, Y) = -\sum_{\text{all } x}\sum_{\text{all } y} p(x, y) \log_2 p(x, y), \tag{10}$$

where $p(x, y)$ is the joint Probability Mass Function (PMF) of X and Y. The condi-

tional entropy of Y given X is

$$H(Y|X) = -\sum_{\text{all } x} p(x) \sum_{\text{all } y} p(y|x) \log_2 p(y|x), \tag{11}$$

where $p(y|x)$ is the conditional PMF of Y given $X = x$.

The mutual information I of two random variables can be defined as the reduction in the uncertainty of one variable, given knowledge of the other. For discrete random variables, I is calculated as

$$\begin{aligned} I(X,Y) &= H(Y) - H(Y|X) & (12) \\ &= \sum_{\text{all } x} \sum_{\text{all } y} p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)} & (13) \end{aligned}$$

where $p(y)$ is the PMF of Y. Now Y provides as much information about X as X does about Y, so

$$\begin{aligned} I(X,Y) &= I(Y,X) & (14) \\ &= H(X) - H(X|Y). & (15) \end{aligned}$$

As an example, suppose X is a random variable drawn from a population of integers uniformly distributed between 1 and 100. Then the entropy $H(X) = \log_2 100 = 6.644$ bits. Now suppose Y is a random variable drawn from a population of integers such that $0 \leq (y - x) \leq 3$ for all $x = X$ and all $y = Y$, and $(Y - X)$ is uniformly distributed between 0 and 3. Then $(y - x)$ can be any of four possible values. If we know x, then it takes $\log_2 4 = 2$ bits of additional information to determine y, so the conditional entropy $H(Y|X) = 2$ bits. It then follows from (12) that the mutual information $I(X,Y) = 6.644 - 2 = 4.644$ bits.

We apply the theory of mutual information to the exact output $\mathcal{G}$, given the esti-

mator $\mathcal{G}'$ computed by the inexact circuit. The mutual information $I(\mathcal{G}, \mathcal{G}')$ provides a useful metric of the quality of $\mathcal{G}'$ as an estimator. The larger the value of I, the better $\mathcal{G}'$ is as an estimator. Note that the value of I is based on an assumed probability distribution of $\mathcal{G}$.

3.1.2 Non-Deterministic.

Non-deterministic error means error caused by random variables unknown to the circuit designer. In conventional circuit design, the goal is to eliminate the uncertainties caused by random errors. However, with inexact computing, non-deterministic error sources may be tolerated. These sources include:

- Thermal noise
- Shot noise
- Flicker $(1/f)$ noise
- Radio Frequency (RF) interference
- Crosstalk within the chip
- Manufacturing process variations
- Radiation-induced single event upsets
- Ionizing radiation effects

3.1.2.1 Analog Systems.

There is no such thing as an error-free analog circuit. Any analog circuit can be thought of as a hardware implementation of a mathematical function $y = f(x)$, where the input vector x and output vector y are both functions of time t. The input nodes of the circuit are corrupted by a noise function $w_{in}(t)$, and the output nodes are corrupted by noise $w_{out}(t)$. By superposition the analog function becomes

$$y(t) = f\left[x(t) + w_{in}(t)\right] + w_{out}(t). \tag{16}$$

Basic building blocks of analog circuit include:

- Amplifier with gain $\mathcal{A}$:

  $y(t) = \mathcal{A} \cdot [x(t) + w_{in}(t)] + w_{out}(t)$

- Adder (summing junction) with n inputs:

  $y(t) = \sum_{i=1}^{n} [x_i(t) + w_{in,i}(t)] + w_{out}(t)$

- Differentiator:

  $y(t) = \frac{\mathrm{d}}{\mathrm{d}t} [x(t) + w_{in}(t)] + w_{out}(t)$

- Integrator with start time t_0:

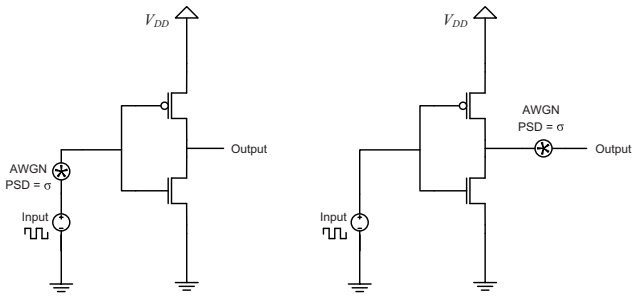
  $y(t) = \int_{t_0}^{t} [x(\tau) + w_{in}(\tau)] \, \mathrm{d}\tau + w_{out}(t)$

In the case of the integrator, the slightest nonzero bias in w_{in} causes error to continuously accumulate with time. In all the above cases, cascading stages of analog circuits introduces additional error with each stage.

By contrast, in traditional computer science, digital logic circuits are considered to be perfectly deterministic, error-free computing machines. Traditional Boolean logic does not consider the possibility of random errors in the system. Inexact computing expands the notion of digital logic to allow errors or approximations to be introduced into the digital system.

3.1.2.2 Binary Logic Affected by Noise.

Schematics for a CMOS inverter with noise at its input and at its output are shown in Fig. 4. We expect the output of an inverter to simply be the binary complement of its input. However, when an inverter is degraded by random noise, its output appears as shown in Fig. 5. This approach can be applied to more complex circuits such as adders, which is the method used in this dissertation.



(a) Noise at input. (b) Noise at output.

Figure 4. CMOS inverter with (a) additive white Gaussian noise (AWGN) at the input, and (b) AWGN at the output.

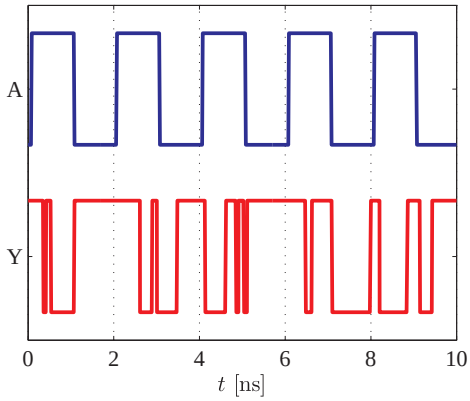


Figure 5. Digitized input waveform A and output waveform Y of a noisy inverter, showing a clean input and a noisy output. In this example, the noise was input-coupled as shown in Fig. 4a.

3.1.2.3 Probabilistic Boolean Logic.

3.1.2.3.1 Definitions.

Chakrapani defines probabilistic Boolean operators, similar to standard Boolean operators, except with a certain probability of correctness [38, 28, 27]:

$\vee_p$ disjunction (OR) with probability p,

$\wedge_q$ conjunction (AND) with probability q, and

$\neg_r$ negation (NOT) with probability r,

where $\frac{1}{2} \leq p, q, r \leq 1$.

The probabilistic equality operator is denoted as:

$\stackrel{s}{=}$ is equal to, with a probability s,

where $\frac{1}{2} \leq s \leq 1$.

The probabilistic AND, OR, and NOT operations described above are very useful for analyzing complex circuits such as adders and multipliers, as described in Sections 4.3-4.4.

3.1.2.3.2 Identities.

In standard Boolean logic, there are several identities which can be extended into probabilistic Boolean logic [38]:

1. Commutativity
2. Double Complementation
3. Operations with 0 and 1
4. Identity
5. Tautology
6. DeMorgan's identities

3.1.2.3.3 Identities which are Not Preserved.

Not all classical Boolean identities can be extended into probabilistic Boolean logic. For binary variables a, b, and c, and probabilities $p_1 \ldots p_5$, the following identities are *not* preserved [38]:

1. Associativity: $(a \vee_{p_1} (b \vee_{p_1} c)) \not\equiv ((a \vee_{p_1} b) \vee_{p_1} c)$.
2. Distributivity: $(a \vee_{p_1} (b \wedge_{p_2} c)) \not\equiv ((a \vee_{p_3} b) \wedge_{p_4} (a \vee_{p_5} c))$.
3. Absorption: $(a \wedge_{p_1} (a \vee_{p_2} b)) \not\equiv a$.

3.1.2.4 Probability of Correctness in the Presence of Random Noise.

Consider a pulsed waveform x which varies with time t. If we add random noise, then the noisy waveform $\tilde{x}$ can be modeled as

$$\tilde{x}(t) = x(t) + w_{in}(t) \tag{17}$$

where w_{in} is Additive White Gaussian Noise (AWGN), expressed as

$$w_{in}(t) = A_w \cos(\omega_{w,x} t + \psi_{w,x}) \tag{18}$$

where the amplitude $A_{w,x}$, frequency $\omega_{w,x}$, and phase $\psi_{w,x}$ are constant within the short sampling window of interest, but otherwise are random variables distributed as

$$A_{w,x} \sim \mathcal{N}\left(0, \sigma_w^2\right) \tag{19}$$

$$\omega_{w,x} \sim \mathbf{Unif}(\omega_L, \omega_U) \tag{20}$$

$$\psi_{w,x} \sim \mathbf{Unif}(0, 2\pi), \tag{21}$$

where ω_L and ω_U are the upper and lower limits of the signal bandwidth, and σ_w is the standard deviation of the noise amplitude. In this dissertation, the notation $\sim \mathcal{N}(\mu, \sigma^2)$ means "is normally distributed with mean μ and variance σ^2", and $\sim$ **Unif**(a, b) means "is uniformly distributed between a and b". In the case of thermal noise, the mean thermal energy is

$$E_x = \tfrac{1}{2} k_B T = \tfrac{1}{2} C_x \langle w_{in}^2(t) \rangle, \tag{22}$$

where k_B is Boltzmann's constant, T is the absolute temperature, C_x is the capacitance of the circuit node, and $\langle w_{in}^2(t) \rangle$ is the mean of the square of the thermal noise voltage over time [29]. Since w_{in} is normally distributed in amplitude, then the average $\langle w_{in}^2(t) \rangle$ is exponentially distributed with mean σ_w^2 [39], which implies a rate parameter σ_w^{-2}. This exponential distribution is denoted as

$$\langle w_{in}^2(t) \rangle \sim \textbf{Exponential}\left(\sigma_w^{-2}\right). \tag{23}$$

Noise sources can create errors in digital circuits. These effects can be simulated using Simulation Program with Integrated Circuit Emphasis (SPICE) or Spectre$^{\text{TM}}$ software tools, as described in Section 4.1.

3.2 Adders

A large part of the JPEG image compression algorithm, as described in Section 3.6, consists of addition and multiplication; and multiplication is built upon addition. A digital adder computes the sum of two N-bit integers A and B to produce an N-bit output "sum" S and a carry-out bit C_{out}. The simplest N-bit adder, the Ripple-Carry Adder (RCA), is composed of N one-bit adders in parallel, with a carry signal connecting each bit to the one above it. In many cases, the sum of A and B

will require $N+1$ bits to store, and therefore S is not always equal to $A+B$. In this research, the sum of A and B is called the augmented sum S^+, where

$$S^+ = A + B \tag{24}$$

and

$$S^+ = 2^N C_{out} + S. \tag{25}$$

The ith bits of A, B, and S are each written as a_i, b_i, and s_i respectively, where

$$A = \sum_{i=0}^{N-1} 2^i a_i, \tag{26}$$

$$B = \sum_{i=0}^{N-1} 2^i b_i, \quad \text{and} \tag{27}$$

$$S = \sum_{i=0}^{N-1} 2^i s_i. \tag{28}$$

For most applications, the ripple-carry adder is the slowest adder architecture. There are many adders which are optimized for speed, for example: carry-lookahead, Brent-Kung, Han-Carlson, and Kogge-Stone adders [40]. Previous work [35] has applied inexactness to these more sophisticated adders. In this research, we will use these types of inexact adders in the image compression algorithm.

3.2.1 1-Bit Full Adder.

The one-bit Full Adder (FA) is the basic building block of a digital adder. It takes three binary inputs a, b, and c_{in} (where c_{in} is known as the carry-in) and produces binary outputs s (the sum) and c_{out} (the carry-out) according to the truth table in Table 1. The circuit diagram for a one-bit full adder is shown in Figure 6.

Table 1. Truth Table for a 1-Bit Full Adder

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 2. Truth Table for a 1-Bit Half Adder

a	b	c_{out}	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

3.2.2 1-Bit Half Adder.

The one-bit Half Adder (HA) is a simplified version of the full adder. The half adder is used when there is no carry-in bit—for example, in the lowest-order bit of an N-bit adder. The truth table for the one-bit half adder is shown in Table 2, and the circuit diagram is shown in Figure 7.

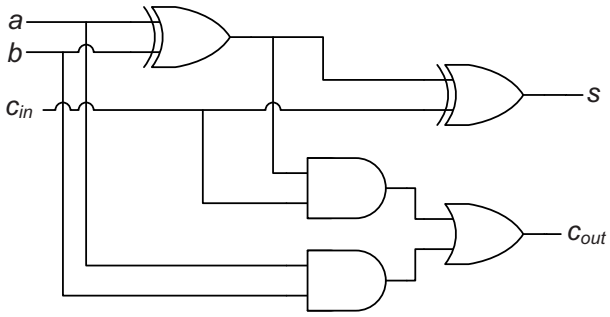


Figure 6. 1-bit full adder

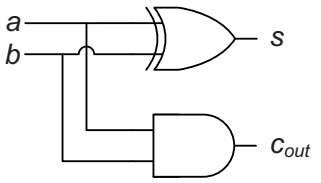


Figure 7. 1-bit half adder

3.2.3 N-bit Ripple-Carry Adder.

An N-bit ripple-carry adder consists of N one-bit adders in parallel, where the ith carry-out bit c_i becomes the carry-in bit to the $(i+1)$th column of the adder. The schematic for an N-bit ripple-carry adder is shown in Figure 8.

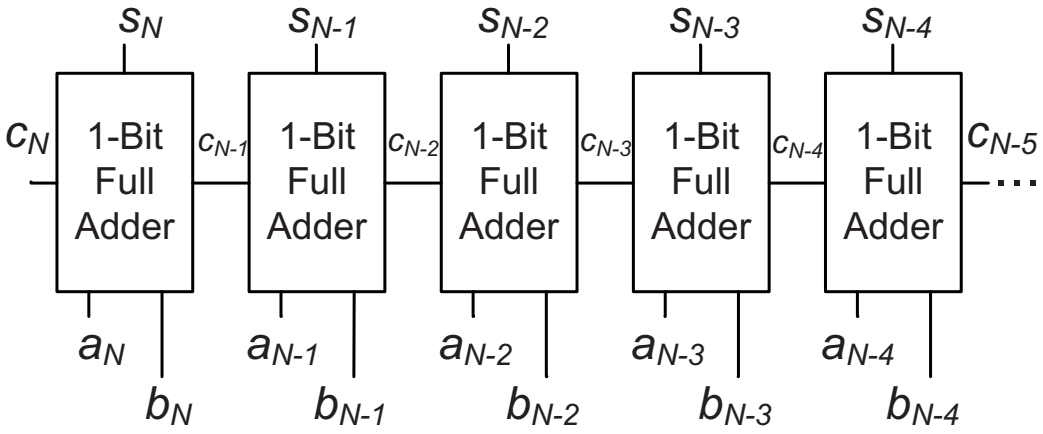


Figure 8. N**-bit ripple-carry adder**

3.2.4 Propagate/Generate Logic.

A carry lookahead adder contains logic designed to quickly compute the higher-value carry bits, so the adder does not have to wait for the carry to ripple from the lowest to the highest bit, as is the case with the ripple-carry adder. This logic is called *propagate* and *generate* logic. A generate condition exists if a column (or group of columns) generates a carry. A propagate condition exists if column(s) propagate a carry which was generated by a lower-value column. The propagate condition $P_{i:j}$ for the jth stage of the ith column is determined by

$$P_{i:j} = P_{i:k} \ \wedge \ P_{k-1:j}, \tag{29}$$

where k represents a previous stage, and $\wedge$ is the logical AND operator. The generate condition $G_{i:j}$ is determined by

$$G_{i:j} = (P_{i:k} \ \wedge \ G_{k-1:j}) \ \vee \ G_{i:k}, \tag{30}$$

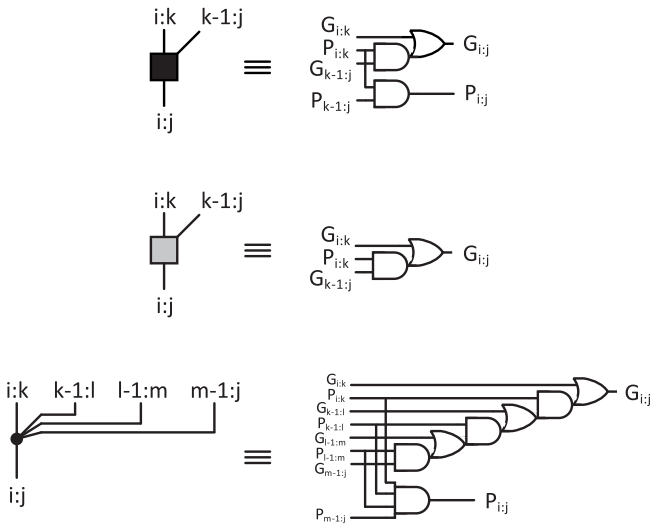


Figure 9. Propagate/Generate logic gates [40].

where $\vee$ is the logical OR operator. The schematic symbols for these expressions are illustrated in Fig. 9.

3.2.5 Ripple-Carry Adder.

A ripple-carry adder is the simplest digital adder, requiring fewer components and less area than any other adder. It is also the slowest type of adder, because it computes every column of the sum one at a time. All other adders have additional components designed to speed up computation by computing the higher-order bits in parallel with the lower-order bits. The schematic for a 16-bit ripple-carry adder is shown in Fig. 10.

3.2.6 Carry Lookahead Adder.

A carry lookahead adder splits the addition problem into subgroups of bits, and has additional hardware, called the carry lookahead logic, which enables fast computation of the higher-order bits. The number of bits in a subgroup is called the *valency* or

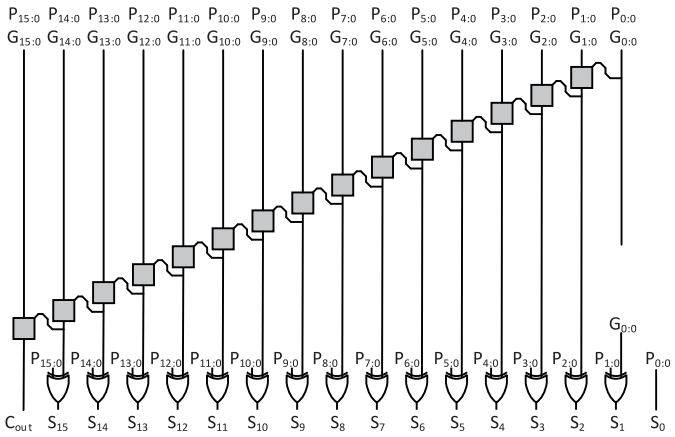


Figure 10. 16-bit ripple-carry adder schematic [40].

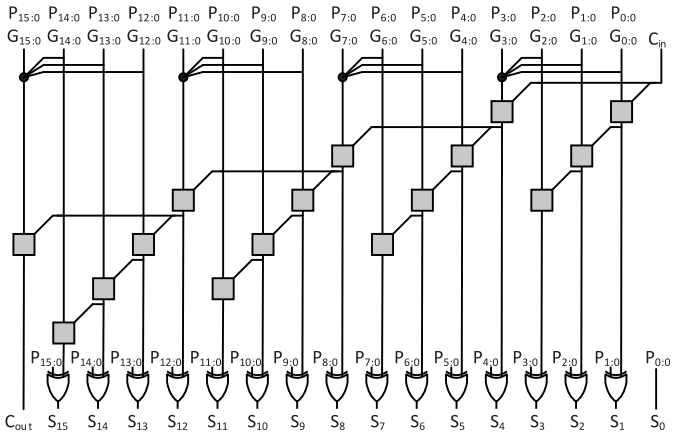


Figure 11. 16-bit radix 4 carry lookahead adder schematic [40].

radix of the adder. For a radix 4 carry lookahead adder,

$$G_{i:j} = G_{i:k} + P_{i:k}\left(G_{k-1:l} + P_{k-1:l}\left(G_{l-1:m} + P_{l-1:m}G_{m-1:j}\right)\right) \tag{31}$$

and

$$P_{i:j} = P_{i:k}P_{k-1:l}P_{l-1:m}P_{m-1:j}, \tag{32}$$

as represented by the black circles in the bottom left of Fig. 9 and the top of Fig. 11 [40]. The schematic for a 16-bit radix 4 carry lookahead adder is shown in Fig. 11.

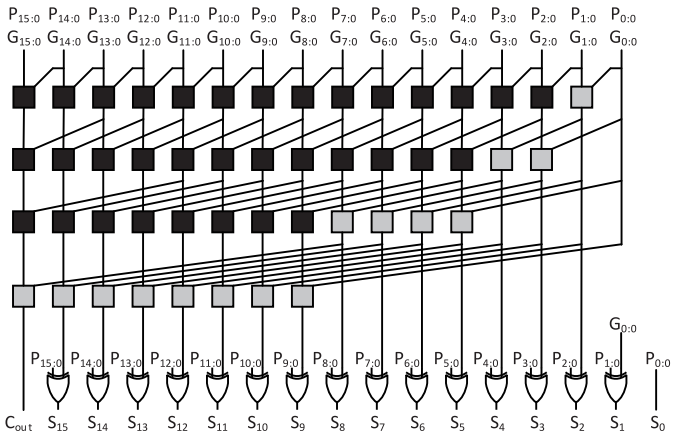


Figure 12. 16-bit Kogge-Stone adder schematic [40].

3.2.7 Kogge-Stone Adder.

The Kogge-Stone adder is a type of carry lookahead adder. In an N-bit Kogge-Stone adder, the carry lookahead logic has $\log_2 N$ stages. The schematic for a 16-bit Kogge-Stone adder is shown in Fig. 12. The inputs to the first stage, $P_{i:0}$ and $G_{i:0}$, are determined by

$$P_{i:0} = a_i \ \oplus \ b_i \tag{33}$$

$$G_{i:0} = a_i \ \wedge \ b_i, \tag{34}$$

where a_i and b_i are the ith bits of the adder inputs A and B, and $\oplus$ is the exclusive-OR operator. The ith sum bit s_i is determined by

$$s_i = P_{i:0} \ \oplus \ G_{i:\log_2 N}. \tag{35}$$

3.2.8 Ling Adder.

A Ling adder [41] uses *pseudogeneate* and *pseudopropagate* signals $H_{i:j}$ and $I_{i:j}$ in place of the regular propagate and generate signals $G_{i:j}$ and $P_{i:j}$ in (29)-(30) and Fig. 9. The Ling adder used in the IBM Power4 microprocessor is a radix 4 carry lookahead adder [42]. Any adder that uses propagate/generate logic can use the Ling technique. The only differences are in the precomputation of $H_{i:0}$ and $I_{i:0}$ at the top of the schematic, and the computation of the final sum bits at the bottom. The initial pseudogenerate is computed as [40]

$$H_{i:0} = a_{i:0}b_{i:0}, \tag{36}$$

and the initial pseudopropagate is

$$I_{i:0} = a_{i:0} + b_{i:0}. \tag{37}$$

The advantage of this is that it replaces an exclusive-OR with an OR on the critical path, which makes the algorithm faster. Then, instead of using (31) to compute the carry lookahead logic, we use the simpler expression

$$H_{i:j} = H_{i:k} + H_{k-1:l} + \bar{K}_{k-1:l}\left(H_{l-1:m} + \bar{K}_{l-1:m}H_{m-1:j}\right), \tag{38}$$

where $\bar{K}_{i:j} = I_{i+1:j+1}$, and instead of (32) we use

$$I_{i:j} = I_{i:k}I_{k-1:l}I_{l-1:m}I_{m-1:j}. \tag{39}$$

The final sum bit S_i is computed as

$$S_i = H_{i-1:0}\left(P_{i:0} \oplus \bar{K}_{i-1:0}\right) + \bar{H}_{i-1:0}P_{i:0}. \tag{40}$$

3.2.9 Probabilistic Boolean Logic (PBL).

In this research, we use PBL to analyze complex circuits such as adders. The probabilistic AND, OR, and NOT operations described in Section 3.1.2.3 are very useful for this purpose, as described in Sections 3.2.10-3.2.11. Although [38] does not define a probabilistic exclusive-or (XOR), for the purpose of this research we define it as

$$a \ \oplus_p \ b = (a \ \wedge_1 \ \neg_1 b) \ \vee_p \ (\neg_1 a \ \wedge_1 \ b) \tag{41}$$

for binary numbers a and b, where $\oplus_p$ is the XOR operation with probability p of correctness.

3.2.10 Propagate/Generate Logic with PBL.

From the perspective of Probabilistic Boolean Logic, (29)-(30) can be modified as follows:

$$\tilde{P}_{i:j} \ = \ \tilde{P}_{i:k} \ \wedge_p \ \tilde{P}_{k-1:j} \tag{42}$$

$$\tilde{G}_{i:j} \ = \ \left(\tilde{P}_{i:k} \ \wedge_1 \ \tilde{G}_{k-1:j}\right) \ \vee_p \ \tilde{G}_{i:k}, \tag{43}$$

where $\tilde{P}_{i:j}$ and $\tilde{G}_{i:j}$ are noisy approximations for $P_{i:j}$ and $G_{i:j}$. In this analysis, the AND-OR-21 (AO21) gate in Fig. 9 is regarded as a single entity, and for this reason the probability p appears only once in (43).

3.2.11 Kogge-Stone Adder with PBL.

PBL can also be applied to the input of the Kogge-Stone adder:

$$\tilde{P}_{i:0} = a_i \ \oplus_p \ b_i \tag{44}$$

$$\tilde{G}_{i:0} = a_i \ \wedge_p \ b_i, \tag{45}$$

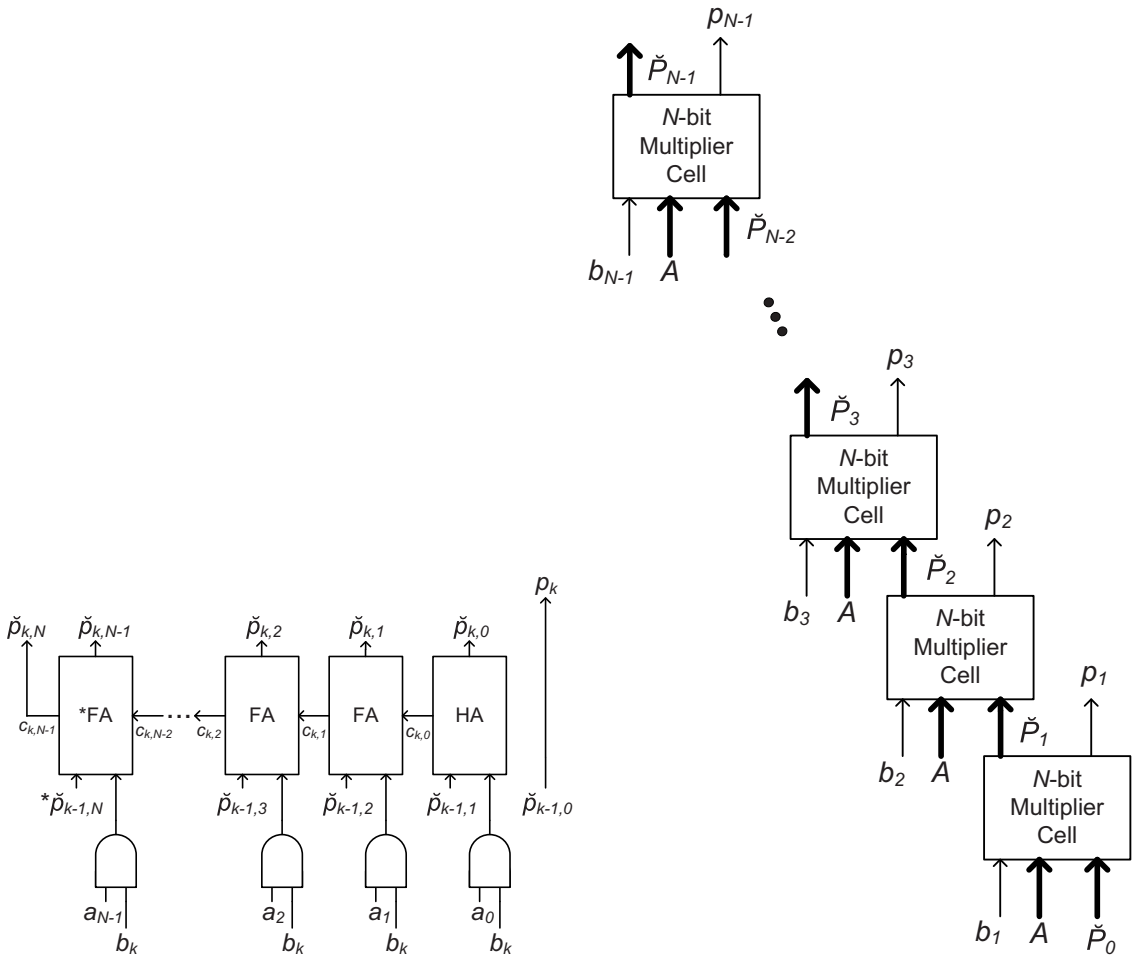


Figure 13. N**-bit integer multiplier. (a) The** k**th stage, not including stage** $k = 0$**. *For stage** $k = 1$**, a half adder can be used for the highest-order bit. (b) All** $(N - 1)$ **stages cascaded together.**

and to the sum bits:

$$\tilde{s}_i = \tilde{P}_{i:0} \ \oplus_p \ \tilde{G}_{i:\log_2 N}, \tag{46}$$

where $\tilde{s}_i$ is a noisy approximation for s_i.

3.3 Multipliers

A multiplier computes the product P of two N-bit integers A and B. For an integer multiplier, P may require up to $2N$ bits to store. In a basic multiplier, the

product is computed as

$$P = 2^N \breve{P}_{N-1} \sum_{k=1}^{N-1} p_k, \tag{47}$$

where $\breve{P}_k$ is the kth partial product, and p_k is the kth bit of P, and

$$p_k = \breve{p}_{k-1,0}, \tag{48}$$

where $\breve{p}_{k-1,0}$ is the zeroth bit of $\breve{P}_{k-1}$. The zeroth partial product $\breve{P}_0$ is computed as

$$\breve{P}_0 = b_0 \sum_{i=0}^{N-1} 2^i a_i, \tag{49}$$

and the remaining $(N-1)$ partial products $\breve{P}_k$ are computed according to

$$\breve{P}_k = \sum_{i=0}^{N-1} \left(a_i b_k + \breve{p}_{k-1,i+1} \right), \tag{50}$$

where $\breve{p}_{k-1,i+1}$ is the $(i+1)$th bit of $\breve{P}_{k-1}$, so

$$\breve{P}_k = \sum_{i=0}^{N} 2^i \, \breve{p}_{k,i}. \tag{51}$$

By Equation (51), $\breve{P}_k$ requires $(N+1)$ bits, and from (47) it is apparent that the final product P of an exact integer multiplier requires $2N$ bits. The schematic for the integer multiplier is shown in Figure 13.

3.4 Probability Distributions

If we treat the inputs A and B to an inexact adder as random variables, it follows that S^+, $\tilde{S}^+$, and $\hat{\varepsilon}$ are also random variables. In this research, we fit the normalized error $\hat{\varepsilon}$ to various hypothetical probability distributions.

3.4.1 Gaussian Distribution.

A Gaussian (normal) distribution is characterized by its Probability Density Function (PDF)

$$f_X(x) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \tag{52}$$

with respect to some random variable X, where μ is the mean and σ is the standard deviation. Given a sample of data, the most likely Gaussian distribution to fit the sample has parameters $\mu = \tilde{\mu}$ and $\sigma = \tilde{\sigma}$, where $\tilde{\mu}$ is the sample mean and $\tilde{\sigma}$ is the sample standard deviation.

3.4.2 Laplacian Distribution.

A Laplacian (double exponential) distribution has the PDF

$$f_X(x) = \frac{\alpha}{2} e^{-\alpha|x|} \tag{53}$$

with scale parameter α and standard deviation $\sqrt{2}/\alpha$ [39]. Given a sample $\mathbf{x}_n$ of data, consisting of n observations $x_1, x_2, \ldots x_n$, the most likely Laplacian distribution to fit the sample has parameter

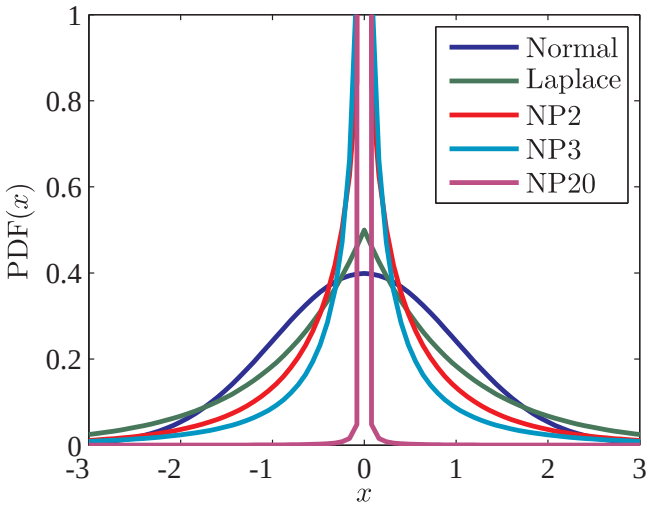
$$\frac{1}{\tilde{\alpha}} = \frac{1}{n} \sum_{j=1}^{n} |x_j - \bar{\mu}|, \tag{54}$$

where $\tilde{\alpha}$ is an estimate of α, and $\bar{\mu}$ is the sample median [43].

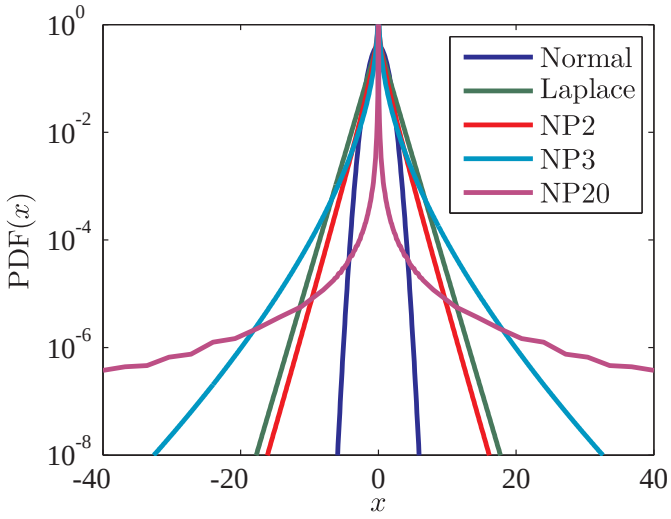
3.4.3 Normal Product Distribution.

A Normal Product (NP) distribution arises from the product u of ψ normally-distributed random variables $X_1, X_2, \ldots X_\psi$, where ψ is the *order* of the distribution, and the product is

$$u = \prod_{j=1}^{\psi} X_j. \tag{55}$$



(a) Linear scale.



(b) Semilogarithmic scale.

Figure 14. Probability density functions for Gaussian (with $\sigma = 1$), Laplacian (with $\alpha = 1$), NP2, NP3, and NP20 (each with $\sigma = 1$).

We introduce the abbreviation NPψ which means "ψth-order normal product distribution". Random variables $X_1, X_2, \ldots X_\psi$ have standard deviations $\sigma_1, \sigma_2, \ldots \sigma_\psi$ respectively, and the product $\sigma = \sigma_1\sigma_2\cdots\sigma_\psi$ is the standard deviation of u. A Gaussian distribution is a special case of a normal product distribution, where $\psi = 1$.

Nadarajah [44] gives the exact formula for the PDF of a normal product distribution as the more general case of the product of ψ Student's t distributions, where ν_i is the number of degrees of freedom of the ith t distribution, and $\nu_i \to \infty$ for all $i \neq i_1, i_2, \ldots i_{\bar{\psi}}$:

$$f_U(u) = \frac{2\pi^{-\psi/2}}{u\Gamma\left(\nu_{i_1}/2\right)\Gamma\left(\nu_{i_2}/2\right)\cdots\Gamma\left(\nu_{i_{\bar{\psi}}}/2\right)} G_{\psi,\bar{\psi}}^{\bar{\psi},\psi} \left(\frac{2^{\psi-\bar{\psi}}\nu_{i_1}\nu_{i_2}\cdots\nu_{i_{\bar{\psi}}}}{u^2} \middle| \begin{matrix} \frac{1}{2}, \frac{1}{2}, \ldots \frac{1}{2} \\ \frac{\nu_{i_1}}{2}, \frac{\nu_{i_2}}{2}, \ldots \frac{\nu_{i_{\bar{\psi}}}}{2} \end{matrix} \right), \quad (56)$$

for $u > 0$, where $\bar{\psi}$ is the total number of non-infinite values of ν_i, Γ is the gamma function, and G is the Meijer G-function. In this research, we are interested in the product of Gaussian distributions, in which case $\bar{\psi} = 0$ and (56) simplifies to

$$f_U(u) = \frac{2\pi^{-\psi/2}}{u} G_{\psi,0}^{0,\psi} \left(\frac{2^\psi}{u^2} \middle| \frac{1}{2}, \frac{1}{2}, \ldots \frac{1}{2} \right). \quad (57)$$

Since the distribution is symmetric about $u = 0$, we can use $|u|$ in place of u. Applying the scale factor σ, the generic formula for a normal product distribution is

$$f_U(u) = \frac{2\pi^{-\psi/2}}{|u|} G_{\psi,0}^{0,\psi} \left(\frac{2^\psi\sigma^2}{u^2} \middle| \frac{1}{2}, \frac{1}{2}, \ldots \frac{1}{2} \right). \quad (58)$$

Fig. 14 compares the PDFs of Gaussian, Laplacian, NP2, NP3, and NP20 distributions. All are symmetric about the central peak. PDFs for all NP distributions with $\psi \geq 2$ diverge to infinity as x approaches zero. The order ψ of an NP distri-

bution is important, because as ψ increases, so does the kurtosis of the distribution. Kurtosis can be interpreted as the narrowness of the peak of the PDF, or alternatively as the heaviness of the tails. This is evident in the NP20 distribution in Fig. 14b, as compared with the NP2 and NP3 distributions.

3.4.4 Maximum Likelihood Estimation.

In order to determine which probability distribution is the best fit for a sample $\mathbf{x}_n$ containing n observations of data, the maximum likelihood estimation method is used. The likelihood l that a sample represents a distribution with PDF f_X is

$$l(\mathbf{x}_n|\theta) = \prod_{j=1}^{n} f_X(x_j|\theta), \tag{59}$$

where θ are the parameters of the distribution. It is usually more convenient to work with the log-likelihood L of the distribution, because this allows us to work with sums instead of products. The log-likelihood is

$$L(\mathbf{x}_n|\theta) = \sum_{j=1}^{n} \ln f_X(x_j|\theta). \tag{60}$$

The parameters θ which maximize l, or equivalently, L, specify the best fit for any given f_X. Furthermore, by the maximum likelihood estimation method it is possible to compare among different distributions, for example, it can determine whether a Gaussian, Laplacian, or normal product distribution is the best fit for the sample. To maximize L, the derivative of L with respect to θ is set to zero:

$$\frac{\partial}{\partial\theta} L(\mathbf{x}_n|\theta) = 0, \tag{61}$$

and the value of θ which satisfies (61) specifies the best fit [39].

According to (58), each normal product distribution PDF for continuous-valued random variables diverges to infinity as U approaches zero. This makes it impossible to perform a maximum likelihood estimation if the data sample contains zeros. This problem can be avoided by treating U as a discrete-valued random variable with bin width $d_{\hat{\varepsilon}}$, where

$$d_{\hat{\varepsilon}} = \frac{1}{2^{N+1} - 1}. \tag{62}$$

Assuming a real discrete-valued U, there is a finite probability that U is equal to some discrete value u:

$$P_U(U = u) = \int_{u - d_{\hat{\varepsilon}}/2}^{u + d_{\hat{\varepsilon}}/2} f_U(\tilde{u})\ \mathrm{d}\tilde{u}. \tag{63}$$

Now P_U can be used in place of f_U in (60) and the maximum likelihood estimation can be performed.

3.5 IEEE 754 Floating Point Storage

The JPEG compression algorithm requires floating point operations in order to perform the Discrete Cosine Transformation (DCT). For floating point numbers, the IEEE 754 standard is the most commonly used method of storage [45]. According to the standard, numbers can be stored within 16, 32, 64, or 128-bit words, and can be stored in base $\mathfrak{b} = 2$ or $\mathfrak{b} = 10$ format. Each floating point number contains three components: a sign bit $\mathfrak{s}$, an exponent $\mathfrak{e}$, and a mantissa $\mathfrak{m}$, where in this notation $1 \leq \mathfrak{m} < 2$. Accordingly, for any floating point number A,

$$A = (-1)^{\mathfrak{s}} \mathfrak{b}^{\mathfrak{e} - \mathfrak{e}_\circ} \mathfrak{m}, \tag{64}$$

Table 3. IEEE 754 Standard Base-2 Formats

Name	Common Name	$\mathfrak{b}$	$\mathfrak{e}_0$	$N_\mathfrak{e}$	$N_\mathfrak{m}$
binary16	Half precision	2	15	5	10+1
binary32	Single precision	2	127	8	23+1
binary64	Double precision	2	1023	11	52+1
binary128	Quadruple precision	2	16383	15	112+1

where $\mathfrak{e}_o$ is the offset bias of the exponent, in accordance with Table 3. Each format includes a $N_\mathfrak{e}$-bit exponent and a $N_\mathfrak{m}$-bit mantissa, where the values for $N_\mathfrak{e}$ and $N_\mathfrak{m}$ are shown in Table 3. For any nonzero binary number, the first digit of $\mathfrak{m}$ is always 1, and is omitted. Therefore, only $(N_\mathfrak{m} - 1)$ bits are stored.

3.5.1 Floating Point Addition.

In order to add two numbers in IEEE 754 format, they must first have the same value for the exponent $\mathfrak{e}$. To obtain this, the mantissa of the smaller number is shifted right, and its exponent incremented, until the exponents of the two numbers match. Trailing bits of the mantissa are truncated. When the two exponents match, then the mantissas can be added. The carry-out bit C_{out} of this addition is then added to the exponent. To find the sum S of two floating point numbers A and B, where $B > A$,

$$\mathfrak{m}_S = \mathfrak{m}_B + (\mathfrak{m}_A >> (\mathfrak{e}_B - \mathfrak{e}_A)) - 2C_{out} \tag{65}$$

$$\mathfrak{e}_S = \mathfrak{e}_B + C_{out}, \tag{66}$$

where $\mathfrak{e}_A$, $\mathfrak{e}_B$, and $\mathfrak{e}_S$ are the exponents of A, B, and S; $\mathfrak{m}_A$, $\mathfrak{m}_B$, and $\mathfrak{m}_S$ are the mantissas of A, B, and S, and $>>$ denotes bitwise shifting by $(\mathfrak{e}_B - \mathfrak{e}_A)$ bits.

3.5.2 Floating Point Multiplication.

Multiplication of two numbers A and B in IEEE 754 format consists of multiplying the mantissas $\mathfrak{m}_A$ and $\mathfrak{m}_B$ together, and then adding the exponents. The leading $N_\mathfrak{m}$

bits of the mantissa are preserved; the rest are truncated. To find the product P,

$$\mathfrak{m}_P = \mathfrak{m}_A \mathfrak{m}_B \tag{67}$$

$$\mathfrak{e}_P = \mathfrak{e}_A + \mathfrak{e}_B, \tag{68}$$

where $\mathfrak{m}_P$ is the mantissa of P, and $\mathfrak{e}_P$ is the exponent of P.

3.6 JPEG Compression Algorithm

The Joint Photographic Experts Group (JPEG) compression algorithm, also known as the JPEG File Interchange Format (JFIF) compression algorithm, consists of the following steps [46, 47]:

1. Color Space Transformation (CST)
2. Tiling
3. Discrete Cosine Transformation (DCT)
4. Quantization
5. Zigzagging
6. Run-amplitude encoding, and
7. Huffman encoding.

These steps are illustrated in the block diagram in Fig. 2.

3.6.1 Color Space Transformation.

First, the image is converted from RGB format to YCbCr format. The reason for this is that the Human Visual System (HVS) is more sensitive to intensity than it

is to color [48]. By converting to YCbCr format, we can optimize the quality of the intensity (luminance) $\mathcal{Y}$ at the expense of the colors (chrominance) $\mathcal{C}_b$ and $\mathcal{C}_r$. The conversion is as follows [49]:

$$\mathcal{Y} = 0.299\mathcal{R} + 0.587\mathcal{G} + 0.114\mathcal{B}; \tag{69}$$

$$\mathcal{C}_b = -0.16874\mathcal{R} - 0.33126\mathcal{G} + 0.5\mathcal{B} + 128; \tag{70}$$

$$\mathcal{C}_r = 0.5\mathcal{R} - 0.41869\mathcal{G} - 0.08131\mathcal{B} + 128. \tag{71}$$

Each component is processed independently of the other components.

3.6.2 Tiling.

Each component ($\mathcal{Y}$, $\mathcal{C}_b$, and $\mathcal{C}_r$) is then arranged into 8×8 tiles of pixels. Each tile is processed independently of the other tiles. The color components are typically sampled at only half the rate of the intensity component in the y direction; that is, the vertical resolution of the color components is half the resolution of the luminance component.

3.6.3 Discrete Cosine Transformation.

Next, the two-dimensional DCT transform is performed on each tile:

$$C = UXU^T, \tag{72}$$

where X is the 8×8 tile of intensity or color data, U is the orthogonal DCT transform matrix, and C is an 8×8 matrix of frequency components in the horizontal and vertical directions. The zero-frequency component is in the upper left corner of the matrix, and is known as the dc component; the others are called the ac components. Assuming the image is a "smooth" function of x and y, most of the DCT components will be

close to zero.

A single 8×8 DCT block produces images as shown in Fig. 15. Each subfigure shows the effect of one of the 64 DCT components being active, with a value of 1023, while all the other components are zero. Not all elements are illustrated in this figure—there are 64 of them. As we will see in Chapter V, if the processor erroneously computes a DCT value which is too large, we will see artifacts which look like the pictures in Fig. 15.

In this research, we simulate probabilistic Boolean logic by randomly flipping bits inside each adder and each multiplier. Matrix multiplication is built from these inexact adders and multipliers. The DCT is built from matrix multiplication. Errors in the DCT result in artifacts like those shown in Fig. 15. Multiple erroneous DCT artifacts may be superimposed onto each other, as well as onto the desired data. An example of erroneous DCT data is shown in Fig. 41f.

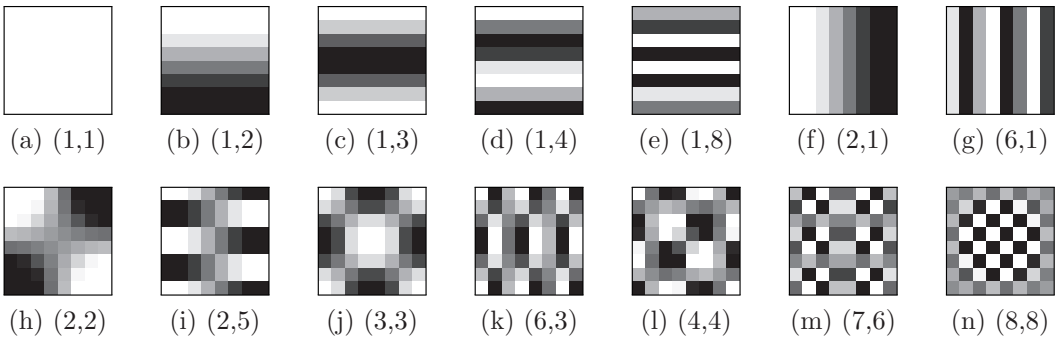


Figure 15. Elementary 8×8 JPEG images, showing the result of a single DCT. In each subfigure, 63 of the 64 values in the DCT matrix are zeros, except for one value which is 1023. The row and column number of the nonzero element is shown in each caption.

In general, finding the product of three 8×8 matrices requires 1,024 multiply operations and 896 addition operations. However, due to the sparseness of the information in U, it is possible to simplify the computational complexity of the DCT. Table 4 shows the complexities of various DCT algorithms. Despite these improvements, computation of the DCT typically takes about 45 percent of the processing

Table 4. Complexity of Various DCT Algorithms for an 8×8 Input Block [48, 47]

Algorithm	Multiplications	Additions	Reference
Block factors	464	144	[50]
2-D Fast Fourier Transform (FFT)	104	474	[51]
Walsh-Hadamard Transform (WHT)	depends on WHT used		[52]
1-D Chen	256	416	[53]
1-D Lee	192	464	[54]
1-D Loeffler, Ligtenberg	176	464	[55]
2-D Kamangar, Rao	128	430	[56]
2-D Cho, Lee	96	466	[57]
1-D Winograd	80	464	[58]

time for the JPEG compression.

3.6.4 Quantization.

The DCT matrix C is then converted to an integer matrix Q via quantization. Matrix C is divided element-wise by a quantization matrix Z and a quantization scale factor α:

$$q_{i,j} = \text{round} \left(\frac{c_{i,j}}{\alpha \, z_{i,j}} \right), \tag{73}$$

where $c_{i,j}$, $z_{i,j}$, and $q_{i,j}$ are the (i,j)th elements of C, Z, and Q respectively. A "quality factor" $\mathfrak{q}$, which is a percentage, is often specified in lieu of α. The scale factor α is calculated from $\mathfrak{q}$ as

$$\alpha = \begin{cases} \frac{50\%}{\mathfrak{q}}, & \mathfrak{q} < 50\% \\ 2 - \frac{\mathfrak{q}}{50\%}, & \mathfrak{q} \geq 50\%. \end{cases} \tag{74}$$

In Equation (73), each $q_{i,j}$ is rounded to the nearest integer. Some information is lost at this point. The quantization matrices are customizable and are saved within the JPEG file; however, standard quantization matrices are commonly used. The intensity (Y) and chrominance (Cb and Cr) components use different quantization

matrices. For the intensity component, the standard quantization matrix is

$$Z = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \quad \text{(for intensity)}, \tag{75}$$

and for the chrominance components, the standard quantization matrix is

$$Z = \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix} \quad \text{(for chrominance)}. \tag{76}$$

From Equation (74), a quality factor $\mathfrak{q} = 50\%$ results in a scale factor $\alpha = 1$, and in that case the unscaled quantization matrices in Equations (75)-(76) are used.

3.6.5 Zigzagging of Q.

Matrix Q is then arranged into a 64-element sequence $\check{Q}$, beginning with the dc component, and zigzagging diagonally from the upper left to the lower right of Q.

3.6.6 Run-Amplitude Encoding.

Run-amplitude encoding is performed on all ac components of $\check{Q}$. These are converted to ordered pairs of integers, where the first number in the pair is the run length of zeros (that is, the number of consecutive zeros in sequence), and the second number is the nonzero component that follows. The last nonzero component of $\check{Q}$ is followed by an End of Block (EOB) code.

3.6.7 Huffman Encoding.

Finally, Huffman encoding is performed on the dc component of $\check{Q}$ and the run-amplitude encoded ac components. The Huffman codes vary in bit length. The value associated with each Huffman code is saved in a table at the beginning of the JPEG file. Optimally, the most frequently occurring Huffman codes have the shortest bit lengths. Usually, four separate Huffman tables are used: dc luminance, ac luminance, dc chrominance, and ac chrominance. These tables are customizable; however, standard Huffman tables are commonly used. Compression occurs during the quantization, run-amplitude encoding, and Huffman encoding steps.

3.6.8 Summary.

The CST, DCT, and quantization involve linear operations (addition and multiplication) which are promising applications for the energy-parsimonious inexact computing described in literature [27]. Several methods exist for optimizing the computational efficiency of the discrete cosine transformation [46, 47]—these could be further improved via inexact computing.

Decoding is accomplished by performing the above seven steps in reverse order. Since losses occur during quantization, the decoded image will not be exactly the same as the original. Also, color information is lost due to the chroma subsampling

described in Section 3.6.2.

Key to the JPEG compression performance is the run-amplitude encoding (Section 3.6.6), which takes advantage of the fact that natural images vary slowly with respect to x and y, have few nonzero ac components, and have long sequences of ac components which are zero. Inexact computing technology must take this into account. Nonzero values interjected into the quantized matrix Q will degrade the compression performance of the system. Single event upsets can also cause this effect.

Since Huffman encoding (Section 3.6.7) and decoding involve variable bit lengths, any error in the code could corrupt all the data that follow. Therefore, the Huffman coding is probably not an application for inexact computing. In a radiation environment, the Huffman codes, like everything else, are susceptible to single event upsets, and could cause such data corruption. Due to the critical nature of the Huffman codes, the Huffman coding algorithm is a good application for hardware or software redundancy, error checking, or radiation hardening.

IV. Methodology

Our approach to characterizing inexact adders consists of a binary probabilistic Boolean logic (PBL) simulation implemented using Matlab. To compute energy and Energy-Delay Product (EDP), we implement adder circuits in a Spectre^{TM} analog simulation. We then calculate the errors of the inexact adders, fit them to probability distributions, and compute summary statistics.

4.1 Circuit Simulations

Noisy analog circuits can be simulated in SPICE or Cadence Spectre^{TM} software via noisy voltage sources, current sources, resistors, or transistors. In the case of voltage or current sources, noise is specified in terms of its Power Spectral Density (PSD) σ^2, measured in volts squared per hertz (V^2/Hz) or amps squared per hertz (A^2/Hz). Thermal noise, shot noise, and flicker noise of resistors and transistors can also be simulated. In all cases, the user must specify the noise bandwidth for the simulation. Noise can also be simulated in Matlab by the addition of a normally distributed random voltage with a mean of zero and a standard deviation σ.

A generalized model for computing the energy consumption, delay, and probability of correctness of an inexact computational circuit, as compared to a conventional (exact) circuit, is shown in Figure 16. Using a SPICE or Spectre^{TM} environment, a Monte Carlo simulation can be performed, using a set X of randomly generated digital input signals which switch at a specified clock rate. This input vector X varies with time t, and is common to both the exact and the inexact circuit. The two circuits are powered by separate voltage sources with magnitude V_{DD} for the exact circuit, and $\tilde{V}_{DD}$ for the inexact circuit, where $\tilde{V}_{DD} \leq V_{DD}$. When the simulation runs, the exact output signal Y and inexact output $\tilde{Y}$ can be then observed and compared.

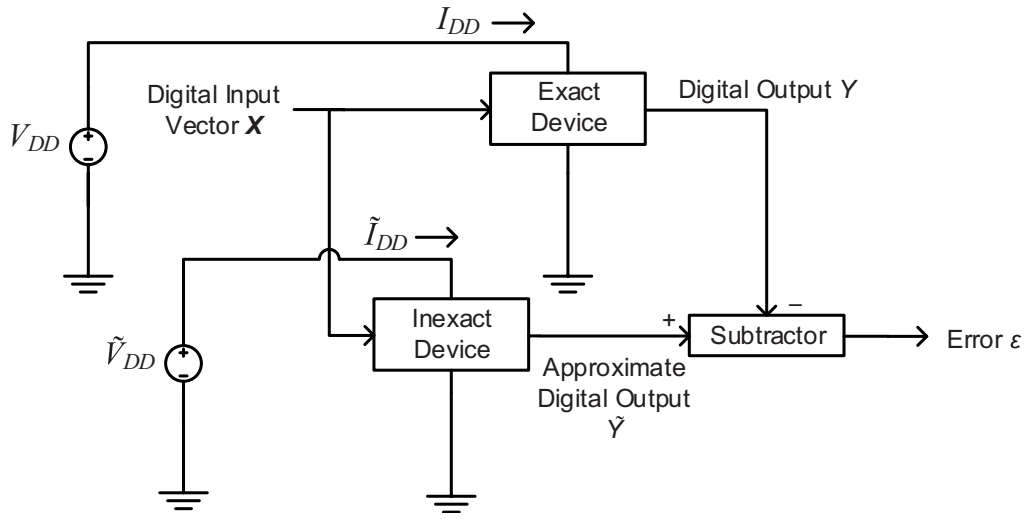


Figure 16. Generalized circuit model for simulating the correctness, delay, and energy consumption of an inexact device, as compared with an exact device.

The difference between the two outputs is the error ε, which is a function of X:

$$\varepsilon = \tilde{Y}\left[X(t)\right] - Y\left[X(t)\right]. \tag{77}$$

If the inexact circuit is noise-susceptible, then Equation 77 becomes

$$\varepsilon = \tilde{Y}\left[X(t)\,, w(t)\right] - Y\left[X(t)\right], \tag{78}$$

where w represents the vector of all noise sources within the inexact circuit.

The simulation also predicts the power supply currents I_{DD} of the exact circuit and $\tilde{I}_{DD}$ of the inexact circuit. Using this information, the instantaneous power $\mathcal{P}$ of the exact circuit can be computed as

$$\mathcal{P}(t) = V_{DD}\, I_{DD}(t), \tag{79}$$

and for the inexact circuit the power is

$$\tilde{\mathcal{P}}(t) = V_{DD}\, \tilde{I}_{DD}(t). \tag{80}$$

The energy consumption $\mathcal{E}$ of the exact circuit is

$$\mathcal{E}(t) = T\, \mathcal{P}(t), \tag{81}$$

where T is the clock period, and for the inexact circuit the energy is

$$\tilde{\mathcal{E}}(t) = T\, \tilde{\mathcal{P}}(t). \tag{82}$$

In digital circuits, the total power $\mathcal{P}$ equals the sum of the dynamic power $\mathcal{P}_d$ and static power $\mathcal{P}_s$. Dynamic power depends on the present input state X_l, and also on the previous state X_{l-1}. Static power depends only on the present input state, and is generally much less than dynamic power. In this research, we are only interested in the average powers $\overline{\mathcal{P}}_d$ and $\overline{\mathcal{P}}_s$ for exact circuits, and average powers $\overline{\tilde{\mathcal{P}}}_d$ and $\overline{\mathcal{P}}_s$ for inexact circuits. While $\mathcal{P}_s$ and $\overline{\mathcal{P}}_s$ are functions of all possible inputs X_l, $\mathcal{P}_d$ and $\overline{\mathcal{P}}_d$ are functions of all possible X_l paired with all possible X_{l-1}. For complex circuits, it is impractical to test the entire input space. For this reason, we choose a random sequence of inputs and perform a Monte Carlo simulation.

4.1.1 Spectre™ Simulation.

4.1.1.1 0.6 μm Technology.

In order to quantify the energy consumption and delay of the inexact 8-bit Kogge-Stone adder, a Monte Carlo simulation of the circuit was conducted in the Cadence Spectre environment, using C5N 0.6 μm technology by ON Semiconductor. Logic

Table 5. Probabilities of Correctness Per Node due to Noise Sources: 0.6 μm Technology

$V_{DD}[V]$	Noise #1 5×10^{-10} V^2/Hz	Noise #2 1×10^{-9} V^2/Hz
1.5	0.9343	0.8603
2.0	0.9773	0.9216
2.5	0.9942	0.9633
3.0	1.0000	0.9831
3.3	1.0000	0.9899

gates were designed for minimum width, with equal pull-up and pull-down strength, and a maximum fanout of 4. Random binary signals $\{a_0, \ldots a_{N-1}, b_0, \ldots b_{N-1}\}$ with a 25% activity factor and a 50 MHz clock rate were used as inputs to the adder. The performance of the adder was observed for 100 cycles (2 μs). Random errors were introduced within the system by placing a Gaussian noise voltage source at each node in the circuit. This effectively simulated the probability p of correctness used in (42)-(46). The noise sources caused each node to vary from its "correct" voltage by a random amount. The probability of correctness was the probability that the noise would not exceed $V_{DD}/2$ at any given point in time. Noise is specified in terms of its bandwidth and Power Spectral Density (PSD). The 0.6 μm simulation used a 500 MHz noise bandwidth and two different PSDs: 5×10^{-10} and 1×10^{-9} V^2/Hz, which produced the per-node probabilities of correctness shown in Table 5. It is worth noting that the noise voltages in Spectre$^{\text{TM}}$ are roughly Gaussian, but were not seen to produce values beyond ± 3.05 standard deviations. It is also worth noting that, although $p = 1$ appears twice in Table 5, this is not necessarily an error-free condition, because the noise could cause additional delay during state transitions, or accumulations of noise from multiple sources could cause errors.

Table 6. Probabilities of Correctness Per Node due to Noise Sources: 14 nm **Technology**

$V_{DD}[V]$	Noise #1 1×10^{-11} V^2/Hz	Noise #2 2×10^{-11} V^2/Hz
0.2	0.6726	0.6241
0.3	0.7488	0.6824
0.4	0.8145	0.7365
0.5	0.8682	0.7854
0.6	0.9101	0.8286
0.7	0.9412	0.8658
0.8	0.9632	0.8970

4.1.1.2 14 nm Technology.

Spectre^{TM} simulations of 8 and 16-bit ripple-carry adders were conducted using a 14 nm finFET technology. Logic gates were simulated using all minimum-width transistors with one finger and two fins. Simulations were run with a 500 MHz clock rate, over a time span of 100 cycles (200 ns). To simulate the noise at each circuit node, two different noise states were used: 1×10^{-11} and 2×10^{-11} V^2/Hz, which produced the per-node probabilities of correctness shown in Table 6. The noise bandwidth in each case was 5 GHz.

4.1.1.3 Energy Per Cycle.

The instantaneous power consumption $P(t)$ of the adder is found by multiplying the power supply current $I_{DD}(t)$ by the power supply voltage V_{DD}:

$$P(t) = V_{DD}\, I_{DD}(t). \tag{83}$$

The average power consumed between times t_1 and t_2 is

$$P_{avg} = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} P(t)\mathrm{d}t, \tag{84}$$

and the average energy per clock cycle is

$$E_{avg} = P_{avg}T = \frac{P_{avg}}{f}, \tag{85}$$

where T is the clock period and f is the clock frequency.

4.1.1.4 Delay and Error.

To measure the normalized error $\hat{\varepsilon}$ of the simulated adder, the exact instantaneous augmented sum $S^+(t)$, as a function of time t, was computed from the input signals according to (24)-(25), and the inexact instantaneous augmented sum $\tilde{S}^+(t)$ was computed from the output signals. The instantaneous error ε is computed as

$$\varepsilon(X, w) = \tilde{S}^+(X, w) - S^+(X), \tag{86}$$

where $X = \{A, B\}$ is the input vector for a two-input adder, w is the vector of noise sources inside the adder, and $\tilde{S}^+$ is the approximate augmented sum computed by the inexact adder. In the analog simulations, each noise source is an additive white Gaussian noise (AWGN) source. For integer adders, it is helpful to normalize the error to its maximum possible value. The normalized error is then

$$\hat{\varepsilon} = \frac{\varepsilon}{\varepsilon_{max}}, \tag{87}$$

where for an N-bit integer adder the maximum possible error is

$$\varepsilon_{max} = \left(2^N - 1\right) + \left(2^N - 1\right) = 2^{N+1} - 2. \tag{88}$$

The instantaneous error $\hat{\varepsilon}(t)$ was computed from (86)-(88), and then interpolated along a uniform spacing of t. Computing error this way is not simple, however. Even

a noise-free adder experiences a time delay δ between the inputs and outputs, and that delay is variable, depending on which input caused the change at the output. In an 8-bit Kogge-Stone adder, the shortest delay δ_{min} is from inputs $\{a_0, b_0\}$ to output s_0, while the longest delay δ_{max} is along the "critical" path, which could be from any of the inputs to outputs s_5, s_6, or s_7. This is evident in Fig. 12. In this figure, we can see that only one gate delay is required to compute $s_0 = P_{0:0} = a_0 \oplus b_0$. To compute s_7, however, one gate delay is required to compute $P_{6:0} = a_6 \oplus b_6$ and $G_{6:0} = a_6 b_6$, and then four more gate delays to compute s_7.

To determine the range of possible delays, a noise-free adder was simulated using power supply voltages $V_{DD} = 1.5,\ 2.0,\ 2.5,\ 3.0$, and 3.3 V. For a given V_{DD}, on each clock pulse, the time span between the minimum and maximum possible delay is considered an indeterminate state. For the purpose of error calculation, the domain of $\hat{\varepsilon}(t)$ was restricted to exclude the indeterminate states. Statistics of the remaining observations of $\hat{\varepsilon}$ could then be calculated. Metrics for the inexact adder are: mean and RMS error, maximum delay δ_{max}, average energy E_{avg}, and the energy-delay product, calculated as

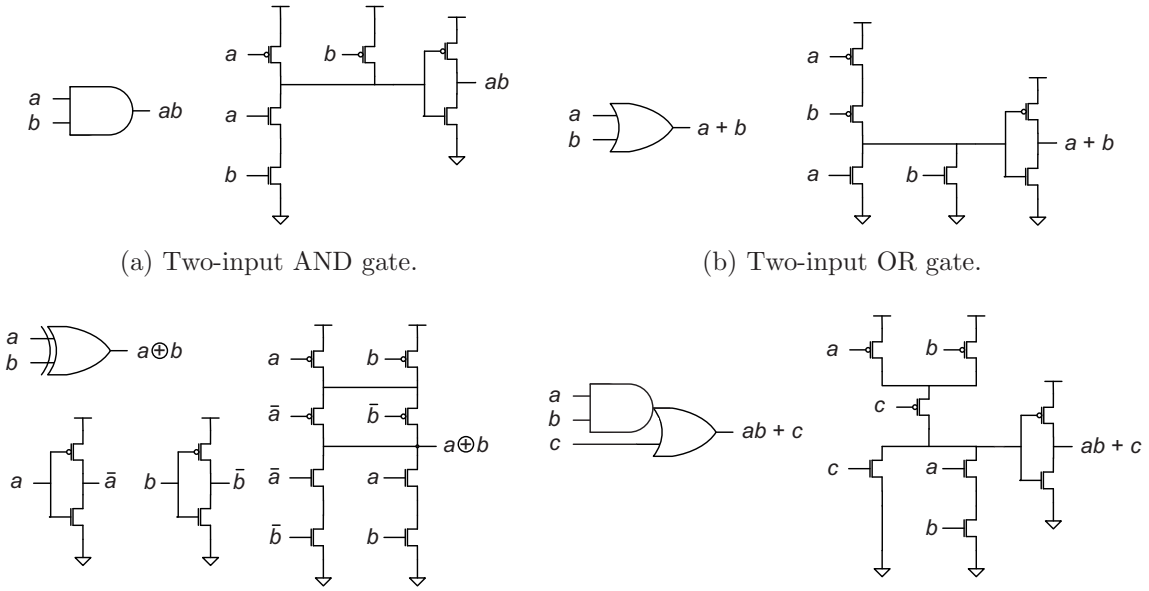
$$\text{EDP} = E_{avg}\, \delta_{max}. \tag{89}$$

4.2 Probabilistic Boolean Logic Simulations

As described in Section 3.2.9, Chakrapani [38] defines the building blocks of inexact digital logic circuits in terms of the probability of correctness p of each individual logic gate. In a SPICE or Spectre$^{\text{TM}}$ environment, a noisy digital logic circuit can be simulated as described in Section 4.1, and at its output we can observe p as a function of σ. In Matlab, a binary circuit node with probability p of correctness can be simulated by comparing p with a uniformly distributed random number, and flipping the node to the "wrong" state if the random number exceeds p. By this

methodology, it is necessary to simulate every node in the circuit. Therefore, when building a complex circuit, it is necessary to have a schematic of the proposed circuit in order to correctly simulate the probabilities of correctness of the final outputs. A basic adder is shown in Figure 8, and a basic multiplier in Figure 13. More complex adders are commonly used in practice, and will be developed for this research.

4.3 Inexact Adders



(a) Two-input AND gate.

(b) Two-input OR gate.

(c) Two-input XOR gate.

(d) AND-OR-2-1 gate.

Figure 17. Schematics for AND, OR, XOR, and AND-OR-2-1 gates.

Many different types of simulations can be performed in order to evaluate the performance of inexact adders:

1. Various adder architectures, including ripple-carry, carry lookahead, Kogge-Stone etc.

2. Various types of inexactness, including probabilistic pruning, probabilistic logic minimization, probabilistic Boolean logic, and noise.

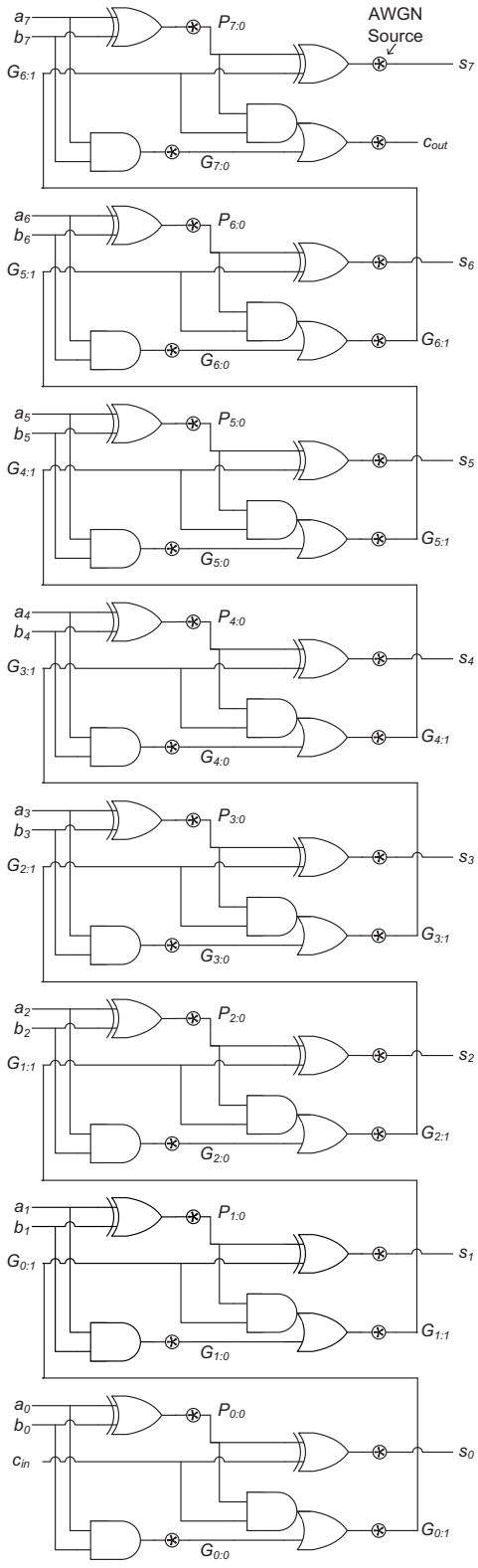


Figure 18. Schematic of a noisy 8-bit ripple-carry adder. Additive white Gaussian noise (AWGN) sources are shown at the output of each AND, XOR, and AND-OR-2-1 gate.

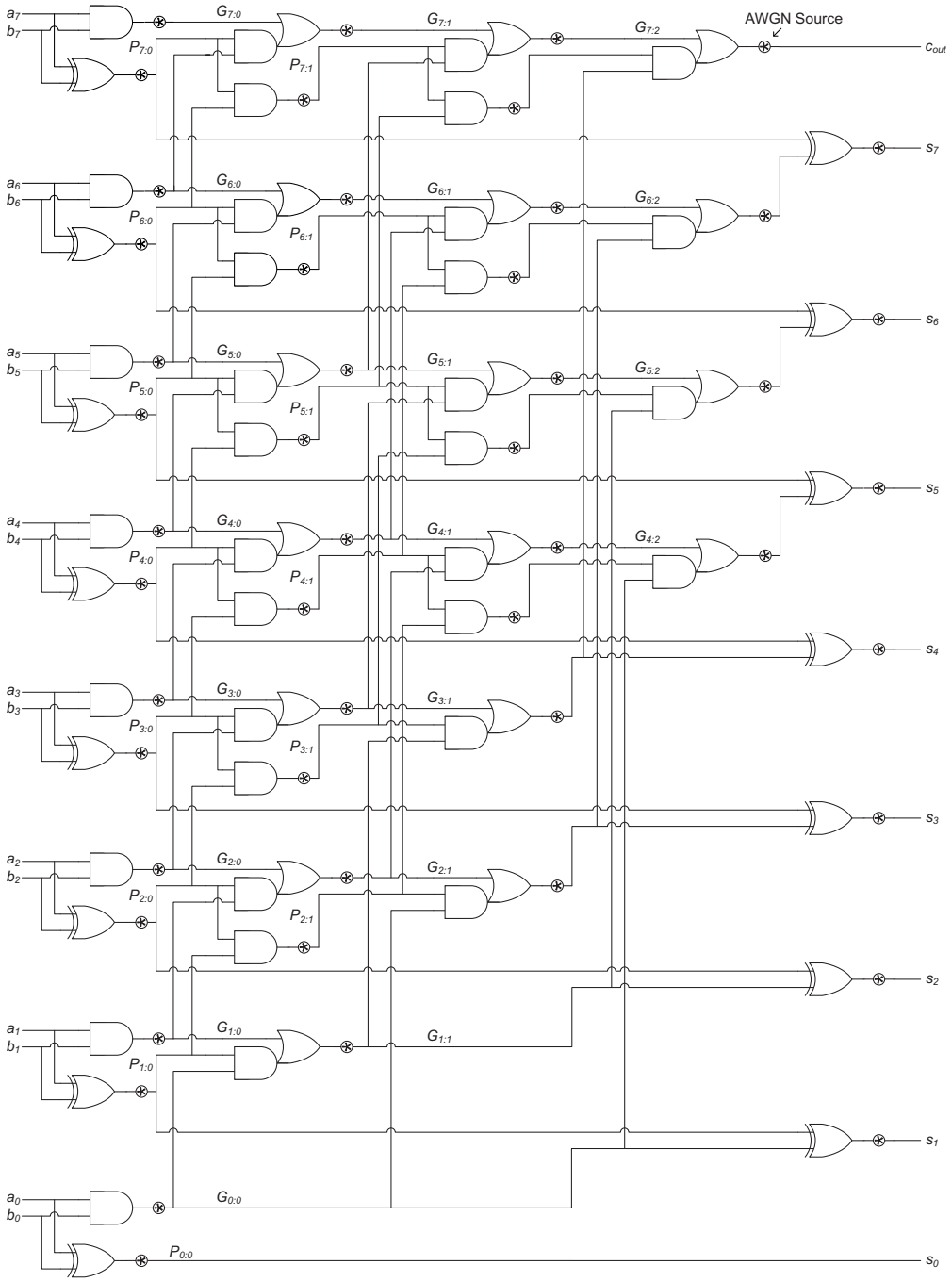


Figure 19. Schematic of a noisy 8-bit Kogge-Stone adder. Additive white Gaussian noise (AWGN) sources are shown at the output of each AND, XOR, and AND-OR-2-1 gate.

3. Various adder sizes: $N = 8,\ 16,\ 32$, or 64 bits.

4. Integer and floating point adders.

5. Various distributions of the inputs A and B.

6. Varying the power supply voltage V_{DD}.

7. Varying amounts of the noise w at each node within the circuit.

8. Varying the probability p of correctness at each node within the circuit.

9. Applying more energy to the higher-order bits, in order to decrease the error ε.

Circuit simulations were performed with Spectre$^{\text{TM}}$ as described in Section 4.1, and probabilistic Boolean logic simulations were performed with Matlab, as described in Section 4.2.

Schematics for the AND, OR, XOR, and AND-OR-2-1 gates are shown in Fig. 17. Positive logic (instead of NAND and NOR) is consistent with [[32], Fig. 11]. The schematic for a noisy 8-bit ripple-carry adder is shown in Fig. 18, and the schematic for a noisy 8-bit Kogge-Stone adder is shown in Fig. 19. In these figures, a noise source is at the output of each logic gate.

Adders were evaluated in terms of energy dissipation, delay, chip area, and error, where the error $\hat{\varepsilon}$ is computed from (86)-(88). In the Spectre$^{\text{TM}}$ analog simulation environment, the noise vector w is a set of additive white Gaussian noise (AWGN) sources located at each node within the circuit. In the Matlab probabilistic Boolean logic (PBL) simulations, the noise sources were discrete-valued, binary error sources. Note that (87)-(88) do not apply to floating-point adders, and it is not practical to normalize errors relative to the upper limits of IEEE 754 floating-point numbers.

4.3.1 Ripple-Carry Adder with Inexactness Only on Less-Significant Bits.

To limit the error of an inexact adder, it is possible to design it such that the most-significant bits are computed using exact (reliable) technology, and the less-significant bits are computed using inexact (unreliable) technology. For an N-bit ripple-carry adder, which is composed of N one-bit adders, the lower $N_{inexact}$ bits would be computed using inexact one-bit adders, and the upper N_{exact} bits would be computed using exact one-bit adders, where $N_{exact} + N_{inexact} = N$. The benefit of this is that for such an adder, ε_{max} is limited to

$$\varepsilon_{max} = \left(2^{N_{inexact}} - 1\right) + \left(2^{N_{inexact}} - 1\right) = 2^{N_{inexact}+1} - 2, \tag{90}$$

and the distribution of the error $\hat{\varepsilon}$ is the same as for an $N_{inexact}$-bit ripple-carry adder. The trade-off is that more energy must be dissipated in order to compute the upper N_{exact} bits. The operation of a partially inexact 8-bit ripple-carry adder with $N_{exact} = 3$ and $N_{inexact} = 5$ is illustrated in Figure 20. In the figure, the five least significant bits, including the carries which ripple upward, are computed using inexact one-bit adders. The three most significant bits, and the carry-out bit, are computed using exact one-bit adders. In this example, it is possible for an erroneous carry bit to ripple from position 4 into position 5; for this reason, it is still possible for the uppermost bits to be erroneous. Repeated addition of such data will cause further accumulation of errors.

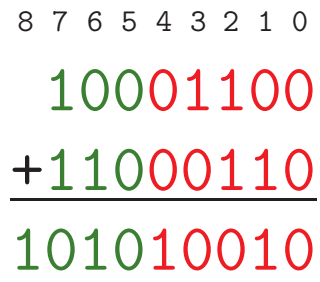


Figure 20. Addition using a partially inexact 8-bit ripple-carry adder with $N_{exact} = 3$ and $N_{inexact} = 5$. Inexact addition is shown in red, and exact addition is shown in green.

4.3.1.1 Energy Savings.

We assume that the energy consumed per cycle by an exact N-bit ripple-carry adder is proportional to N:

$$E_{add,ex} = NE_{1bit-add,ex}, \tag{91}$$

where $E_{1bit-add,ex}$ is the energy per cycle consumed by an exact one-bit adder. The energy consumed by a partially inexact adder is

$$E_{add,in} = N_{exact}E_{1bit-add,ex} + N_{inexact}E_{1bit-add,in}, \tag{92}$$

where $E_{1bit-add,in}$ is the energy per cycle consumed by an inexact one-bit adder. The energy savings of the partially inexact adder relative to the exact adder is

$$\frac{E_{add,in}}{E_{add,ex}} = \frac{N_{exact}E_{1bit-add,ex} + N_{inexact}E_{1bit-add,in}}{NE_{1bit-add,ex}}. \tag{93}$$

4.4 Inexact Multipliers

A methodology similar to Section 4.3 was applied to multipliers. For this research, shift-and-add and Wallace tree architectures were studied. For a multiplier, the error

is computed as

$$\varepsilon(X, w) = \tilde{P}(X, w) - P(X), \tag{94}$$

where P is the exact product and $\tilde{P}$ is the inexact product. For integer multipliers, the normalized error $\hat{\varepsilon}$ is computed using the maximum possible error ε_{max}, which is

$$\varepsilon_{max} = \left(2_P^N - 1\right) \cdot \left(2_P^N - 1\right) = 2^{2N_P} - 2^{N_P+1} + 1, \tag{95}$$

where N_P is the number of bits of the output product of the multiplier, and

$$N_P = N_A + N_B, \tag{96}$$

where N_A and N_B are the bit widths of the multiplicands A and B respectively.

4.4.1 Shift-and-Add Multiplier with Inexactness Only on Less-Significant Bits.

To limit the error of the shift-and-add multiplier, exact technology can be used to compute the most significant bits, with inexact technology used for the less significant bits. The schematic for a shift-and-add multiplier is shown in Figure 13. In this work, we simulate an N-bit partially inexact multiplier as follows. We are given an N_A-bit multiplicand A and an N_B-bit multiplicand B, where $N_A + N_B \leq N$. The uppermost N_{exact} bits are computed exactly, and the remaining $N_{inexact}$ bits are computed inexactly, where $N_{exact} + N_{inexact} = N$. The steps involved in shift-and-add multiplication are:

1. A logical AND is performed between each bit of A and bit 0 (the least significant bit) of B. The AND operation is inexact for bits 0 through $N_{inexact} - 1$; for higher bits, the AND operation is exact. The resulting value is called $\breve{P}_0$.

2. A logical AND is performed between each bit of A and bit 1 of B. The AND operation is inexact for bits 1 through $N_{inexact} - 1$; for higher bits, the AND operation is exact.

3. The value from step 2 is shifted one place to the left.

4. The value from step 3 is added to the value from step 1. In this work, the addition is performed by an N_A-bit ripple-carry adder with a half-adder on the LSB. The addition is inexact for bits 1 through $N_{inexact} - 1$; for higher bits, the addition is exact. No operation is required on the LSB of $\breve{P}_0$; it is simply routed into the bit 0 position of the sum. The result is an $(N_A + 2)$-bit value $\breve{P}_1$.

5. Steps 2 through 4 repeat for each of the bits in B, with the number of shifts in step 3 incrementing with each iteration. However, inexact AND and inexact addition are performed only on bit positions $N_{inexact} - 1$ and below.

An example of partially inexact multiplication is illustrated in Figure 21. In this figure, $N_A = 4, N_B = 4, N_{exact} = 3$, and $N_{inexact} = 5$. The figure illustrates inexact computation performed on the lower five bits of the product.

4.4.1.1 Energy Savings.

We now derive the formula for the energy savings of an inexact multiplier. The energy per cycle consumed by an exact shift-and-add multiplier can be computed as

$$E_{mult,ex} = N_A N_B E_{and,ex} + N_A(N_B - 1)E_{1bit-add,ex}, \tag{97}$$

where $E_{and,ex}$ is the energy per cycle consumed by an exact AND gate, and $E_{1bit-add,ex}$ is the energy per cycle consumed by an exact one-bit adder. For this research, Eq. (97) provides the baseline against which the energy cosumption of inexact multipliers

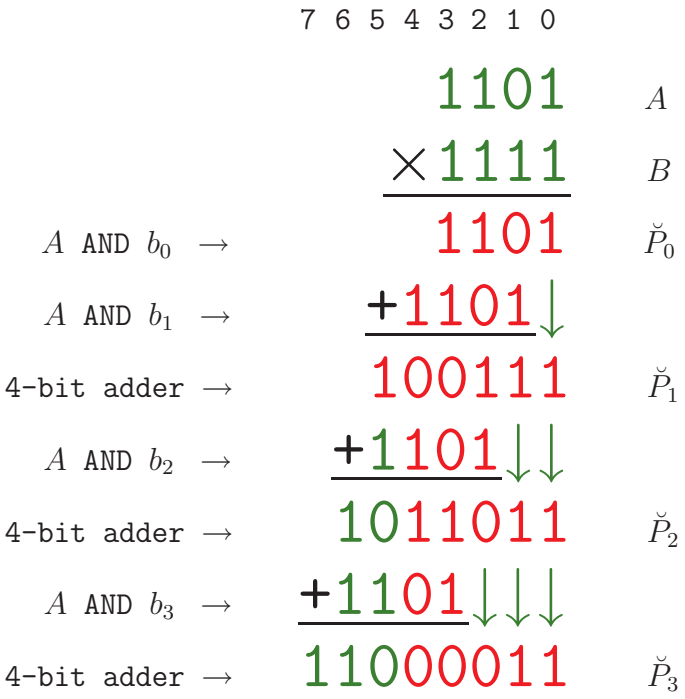


Figure 21. Multiplication using a partially inexact 8-bit shift-and-add multiplier with $N_{exact} = 3$ and $N_{inexact} = 5$. Inexact AND and inexact addition are shown in red, and exact AND and exact addition are shown in green. In this example, the leftmost two bits (sum and carry-out) of $\breve{P}_1$ are the output of a single inexact one-bit adder. Likewise, the leftmost two bits of $\breve{P}_2$ and $\breve{P}_3$ are each output from a single exact one-bit adder.

is compared. In this research, we let $N_{mult,ex} \leq N_B$, where $N_{mult,ex}$ is the number of uppermost bits in the final product P which are to be computed exactly. Given that constraint, from the lower left portion of Fig. 21 we can see that the number of error-free AND gates is

$$N_{mult,and,ex} = \sum_{k=1}^{N_{mult,ex}-1} k = \frac{(N_{mult,ex}-1)N_{mult,ex}}{2}, \tag{98}$$

and the number of exact one-bit adders is also

$$N_{mult,add,ex} = \sum_{k=1}^{N_{mult,ex}-1} k = \frac{(N_{mult,ex}-1)N_{mult,ex}}{2} \tag{99}$$

$$= N_{mult,and,ex}. \tag{100}$$

The number of inexact AND gates is

$$N_{mult,and,in} = N_A N_B - N_{mult,and,ex}, \tag{101}$$

and the number of inexact one-bit adders is

$$N_{mult,add,in} = N_A(N_B - 1) - N_{mult,add,ex}. \tag{102}$$

Table 7 shows the values of $N_{mult,and,ex}, N_{mult,and,in}, N_{mult,add,ex}$, and $N_{mult,add,in}$ computed using Equations (98)-(102), for 16-bit adders with values of $N_{mult,ex}$ ranging from 0 to 6 bits. For example, if the final output product P is to have its three most significant bits computed exactly, i.e. $N_{mult,ex} = 3$, then among the internal components of the multiplier, the number of exact AND gates $N_{mult,and,ex} = 3$, the number of inexact AND gates $N_{mult,and,in} = 61$, the number of exact one-bit adders $N_{mult,add,ex} = 3$, and the number of inexact one-bit adders $N_{mult,add,in} = 53$. The table

shows how the total number of exact ANDs and exact additions increase as $N_{mult,ex}$ increases, and the total number of inexact ANDs and inexact additions decrease as $N_{mult,ex}$ increases; that is, as $N_{mult,ex}$ increases from 0 to 6, $N_{mult,and,ex}$ and $N_{mult,add,ex}$ each increase from 0 to 15, while $N_{mult,and,in}$ decreases from 64 to 49 and $N_{mult,add,ex}$ decreases from 56 to 41. In this example, where $N_A = 8$ and $N_B = 8$, the table shows that in every row, $N_{mult,and,ex} + N_{mult,and,in} = 64$ (from Eq. (101)) and $N_{mult,add,ex} + N_{mult,add,in} = 56$ (from Eq. (102)). Table 7 shows that, based on the model defined in Equations (98)-(102), a multiplier cannot have a parameter $N_{mult,ex} = 1$, since the two upper bits of the final product are the carry-out and sum bits of a single one-bit adder, and we have chosen to define a one-bit adder as containing either all exact components or all inexact components.

Table 7. 16-Bit Shift-and-Add Multipliers Using Exact & Inexact Bits
$N_A = 8$, $N_B = 8$

$N_{mult,ex}$	$N_{mult,and,ex}$	$N_{mult,and,in}$	$N_{mult,add,ex}$	$N_{mult,add,in}$
0	0	64	0	56
1	0	64	0	56
2	1	63	1	55
3	3	61	3	53
4	6	58	6	50
5	10	54	10	46
6	15	49	15	41

Assuming that the energy consumption due to inexact AND is proportional to the number of bits computed that way, then the energy consumption of AND gates is

$$E_{mult,and,in} = N_{mult,and,in} E_{and,in}, \tag{103}$$

where $E_{and,in}$ is the per-cycle energy consumption of a single AND gate. Assuming that the energy consumption due to inexact addition is proportional to the number

of bits so computed,

$$E_{mult,add,in} = N_{mult,add,in} E_{1bit-add,in}, \tag{104}$$

where $E_{1bit-add,in}$ is the per-cycle energy consumption of a one-bit adder. The energy consumption of the partially inexact multiplier relative to the exact multiplier is

$$\frac{E_{mult,in}}{E_{mult,ex}} = \frac{E_{mult,and,in} + E_{mult,add,in}}{E_{mult,and,ex} + E_{mult,add,ex}} \tag{105}$$

$$= (N_{mult,and,in} E_{and,in} + N_{mult,and,ex} E_{and,ex} + N_{mult,add,in} E_{1bit-add,in} + N_{mult,add,ex} E_{1bit-add,ex}) \times \frac{1}{(N_{mult,and,in} + N_{mult,and,ex}) E_{and,ex} + (N_{mult,add,in} + N_{mult,add,ex}) E_{1bit-add,ex}} \tag{106}$$

From Eq. (101)-(102) we can assume $N_{mult,and,in} \not\gg N_{mult,add,in}$. Since a one-bit adder contains two XOR gates, two AND gates, and one OR gate (as shown in Fig. 6), we can assume based on this component count that the energy consumption of a one-bit adder is much greater than that of a single AND gate, that is, $E_{1bit-add,in} \gg E_{and,in}$, and $E_{1bit-add,ex} \gg E_{and,ex}$. Using these assumptions, Eq. (106) simplifies to

$$\frac{E_{mult,in}}{E_{mult,ex}} \approx \frac{N_{mult,add,in} E_{1bit-add,in} + N_{mult,add,ex} E_{1bit-add,ex}}{(N_{mult,add,in} + N_{mult,add,ex})\, E_{1bit-add,ex}}. \tag{107}$$

4.5 Distribution Fitting

Given n observations of the normalized error $\hat{\varepsilon}$, it is desirable to characterize the sample in terms of a probability distribution. A Gaussian fit can be obtained by calculating the sample mean and sample standard deviation. A Laplacian fit can be obtained using (54). For normal product distributions with small values of ψ, the PDF (58) and the data sample can be applied to (60) to determine the value of $\tilde{\sigma}$

Figure 3. Original uncompressed image of an F-16, file name 4.2.05.tiff, from the USC SIPI image database [25]. (repeated from page 15)

which maximizes the log-likelihood, where $\tilde{\sigma}$ is an estimate of the scale parameter σ.

For higher-order normal product distributions, (58) becomes intractable. However, it is possible to simulate normally-distributed variables $X_1, X_2, \ldots X_\psi$ using a random number generator, and multiply them together in accordance with (55) to obtain a random sample of numbers from the desired population. From a random sample, a histogram can be generated, and from this an empirical PDF can be inferred. The empirical PDF can then be used with (60) to find the estimated distribution parameter $\tilde{\sigma}$. This was accomplished for NP distributions with $4 \leq \psi \leq 40$ using a sample size of 10^8.

4.6 Optimizing the JPEG Algorithm for Inexact Computing

The preceding sections have described our methodology for simulating inexact adders and multipliers. This section describes our methodology for limiting the precision of the computations within the JPEG compression algorithm, consistent with Section 3.1.1.1, and also our choice to use exact computation of on the most significant

bits of each addition and multiplication operation. The image compression analysis in this dissertation was performed using the data set shown in Fig. 3, from the University of Southern California (USC) Signal and Image Processing Institute (SIPI) image database [25]. The still images in the SIPI database are all uncompressed tagged image file format (TIFF) files.

4.6.1 Limited Precision.

The inexact computing decision flowchart in Fig. 1 indicates that for purposes of saving energy, delay, and area, we should not use any more precision than is necessary to store data. The JPEG compression algorithm is described in Section 3.6 and illustrated in Fig. 2. In (72) we have two 8×8 matrix multiplications: U times Y, and then $[UY]$ times U^T. Y is an 8-bit signed integer ranging from -128 to 127. The values in U range from -0.49 to $+0.49$. Although these are fractional numbers, we can use integer multiplication to perform the DCT: we can multiply each element of U by $2^{\Delta N_U}$, where ΔN_U is a positive integer, and for negative elements in U we use the two's complement representation. This method is valid as long as we divide by $2^{\Delta N_U}$ after the DCT is complete. In this work, we let $\Delta N_U = 7$, so U becomes an 8-bit signed integer (that is, one sign bit followed by a 7-bit significand). Multiplying one element of U by one element of Y results in a 15-bit signed product (one sign bit followed by a 7+7=14-bit significand). However, an additional three bits are needed to accomplish an 8×8 matrix multiplication. The (r, c)th element of $[UY]$, denoted $[UY]_{r,c}$, is computed by

$$[UY]_{r,c} = \sum_{k=1}^{8} U_{r,k} Y_{k,c} \tag{108}$$

$$= U_{r,1}Y_{1,c} + U_{r,2}Y_{2,c} + U_{r,3}Y_{3,c} + U_{r,4}Y_{4,c} + U_{r,5}Y_{5,c} + U_{r,6}Y_{6,c} + U_{r,7}Y_{7,c} + U_{r,8}Y_{8,c}. \tag{109}$$

To understand the need for three additional bits, consider each pair of terms within Eq. (109), where $(U_{r,1}Y_{1,c} + U_{r,2}Y_{2,c})$ is the first pair, $(U_{r,3}Y_{3,c} + U_{r,4}Y_{4,c})$ is the second pair etc. Each term consists of a 15-bit signed product. Adding a pair of them together (with carry-out) produces a 16-bit signed sum, which can be up to twice as large as a single term. If we group the terms into quadruples (or pairs of pairs), the sum can be up to twice as large as a pair, requiring another bit. If we view the eight terms of Eq. (109) as a pair of quadruples, the sum can be up to twice as large as a quadruple, which requires another bit. So in our example, to perform an 8×8 matrix multiplication it requires four 15-bit additions, two 16-bit additions, and one 17-bit addition, producing an N_{UY}-bit signed product, where $N_{UY} = 18$. When performing the DCT, we then multiply $[UY]$ by U^T. This adds another 7 bits due to the multiplication and another 3 bits due to the addition, for a 28-bit signed product. To properly scale the final DCT, we shift it 14 places to the right. This leaves an 8×8 matrix of N_{UYU^T}-bit signed integers, each of which can store a range of possible DCT values from -8192 to $+8191$, where $N_{UYU^T} = 14$.

However, we do not need that much precision and, as explained in Section 3.1.1.1, precision costs energy, delay, and area. Recall that the original image data contains only 8 bits of information per pixel. Without much loss of fidelity, after computing the intermediate product $[UY]$, we can truncate the lower 8 bits, leaving only a 10-bit signed representation of $[UY]$. If we drop 8 bits from the intermediate product, then we only drop 6 bits from the final product.

Furthermore, a DCT range from -8192 to $+8191$ is not representative of realistic image data. JPEG is designed for photographs of the natural environment, and that kind of data usually varies slowly throughout space, resulting in smaller DCT values, especially in the lower right corner of the DCT matrix. Analysis of the "Mandrill" (4.2.03.tiff), "Lena" (4.2.04.tiff), "F-16" (4.2.05.tiff), and "San Francisco" (2.2.15.tiff)

images from the SIPI database [25] reveals that for each element in $[UY]$, fewer than 10 bits are needed to represent the data, and for the final DCT $[UYU^T]$, fewer than 14 bits are needed. This is especially true for the lower right corner of the matrix, representing high-frequency components which are usually small. Different bit widths are appropriate for different positions within the DCT matrix. In this research, the following bit widths were used:

$$N_{UY} = \begin{bmatrix} 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 7 & 7 & 7 & 8 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 7 & 7 & 6 & 7 & 7 & 7 & 6 & 6 \\ 7 & 7 & 6 & 6 & 7 & 6 & 6 & 6 \end{bmatrix} \tag{110}$$

$$N_{UYU^T} = \begin{bmatrix} 11 & 11 & 10 & 10 & 9 & 9 & 9 & 9 \\ 11 & 10 & 10 & 9 & 9 & 9 & 9 & 9 \\ 10 & 10 & 9 & 9 & 9 & 8 & 9 & 8 \\ 10 & 9 & 9 & 9 & 9 & 9 & 8 & 8 \\ 9 & 9 & 9 & 8 & 9 & 8 & 8 & 8 \\ 9 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 8 & 7 & 8 & 7 \\ 8 & 8 & 8 & 7 & 8 & 7 & 7 & 7 \end{bmatrix}. \tag{111}$$

These numbers are data-dependent. We assume our DCT data can fit into the bit widths in Eq. (110)-(111). If any DCT data overflow beyond these bit widths, errors will occur in the affected 8×8 blocks. In that case, it would be necessary to modify

Eq. (110)-(111) to accommodate the necessary range of DCT values. However, we do not want to use larger bit widths than necessary, because that consumes more energy, and when using inexact components, it risks introducing more errors into the DCT computations.

4.6.2 Exact Computation of the Most Significant Bits.

It is our experience that inexact addition and multiplication of the most significant bit produce unacceptably large and frequent errors in the CST and DCT stages of the JPEG algorithm. In the CST, "unacceptable" errors result in pictures with a lot of "bad" pixels which are either extremely dark or extremely bright. In the DCT, errors manifest themselves in 8×8 blocks, and pictures with "unacceptable" errors have numerous and intense artifacts as shown in Fig. 15. To limit the errors, we use exact computation on the three most significant bits of every addition and multiplication operation, as explained in Sections 4.3.1 and 4.4.1. We also use exact computation when computing two's complement, and when computing the sign of a multiplication operation.

4.7 JPEG Compression Performance

In this research, we divide the JPEG compression algorithm into its sub-algorithms, as shown in Fig. 2, and then examine each one to determine which sub-algorithms are candidates for inexact design, as shown in the flowchart in Fig. 1. The color space transformation, discrete cosine transformation, and quantization sub-algorithms are computational in nature, and are therefore candidates for inexact design. Tiling and zigzagging are routing processes, and run-amplitude encoding and Huffman encoding are encoding processes; these are not candidates for inexact design. The discrete cosine transform is the most computationally intensive, and would most likely require

the largest chip area and energy consumption of all the sub-algorithms. The choices of RMS error, signal-to-noise ratio, and compression ratio are examples of choosing error metrics as shown in the flowchart.

This research views JPEG compression performance in terms of image distortion and compression ratio. Image distortion is a subjective concept which depends on the opinion of a human observer. Without resorting to psychovisual experiments, the most commonly used measures of distortion are the Mean Square Error (MSE) and SNR of the JPEG image which has been encoded and decoded, as compared to the original source image [59]. Mean Square Error is computed as

$$\text{MSE} \quad = \quad \frac{1}{N_{pix,x} N_{pix,y}} \sum_{j=1}^{N_{pix,y}} \sum_{i=1}^{N_{pix,x}} \left(Y'_{i,j} - Y_{i,j} \right)^2 , \tag{112}$$

where $Y_{i,j}$ is the original (i,j)th pixel value, $Y'_{i,j}$ is the decoded pixel value from the JPEG file, $N_{pix,x}$ is the number of pixel columns, and $N_{pix,y}$ is the number of pixel rows. The RMS error is the square root of the MSE, and the SNR is

$$\text{SNR} \quad = \quad 10 \ \log \ \frac{\max \left(Y_{1,1} \ldots Y_{N_{pix,x},N_{pix,y}} \right) - \min \left(Y_{1,1} \ldots Y_{N_{pix,x},N_{pix,y}} \right)}{\text{RMS error}} \quad \text{[dB]}. \tag{113}$$

In (113), the difference between the maximum and minimum pixel values is usually close to 255. Since the pixel intensity has a range of 256 possible values, the relative RMS error is simply the RMS error normalized to 256:

$$\text{Relative RMS Error} = \frac{\text{RMS Error}}{256}. \tag{114}$$

Compression ratio is the ratio between the original file size (in bits) to the size of the encoded file (in bits). For a grayscale image, each pixel of the original image is

represented by 8 bits, and the compression ratio R is

$$R = \frac{8N_{pix,x}N_{pix,y}\ [\text{bits}]}{\text{compressed file size [bits]}} \quad [\text{bits per bit}]. \tag{115}$$

For a three-component color image, the compression ratio is

$$R = \frac{24N_{pix,x}N_{pix,y}\ [\text{bits}]}{\text{compressed file size [bits]}} \quad [\text{bits per bit}]. \tag{116}$$

This metric can also be expressed in terms of bits per pixel; that is, $8/R$ bits per pixel for a grayscale image, and $24/R$ bits per pixel for a color image. The compression ratio is heavily dependent on the amount of quantization performed, which in turn depends on the quality factor α described in Section 3.6.4. Given a quality factor $\alpha = 0.25$, a typical compression ratio $R = 22$ with RMS error equal to 4.8 [46].

4.8 Matlab Scripts

Monte Carlo simulations of adders and multipliers using PBL-based error models can run very fast in Matlab, enabling data collection with large sample sizes. For example, using a desktop computer, Matlab can simulate a 16-bit Kogge-Stone adder 10^6 times within a few seconds. More realistic simulations of adders and multipliers can be performed using Spectre$^{\text{TM}}$ as described in Section 4.1; however, the Matlab PBL simulations can process much more data within a short period of time.

This section presents the function calls to the Matlab-based circuit simulations. These functions were designed to run using Matlab version 2015a. They correspond to the results shown in Chapter V. The complete functions are given in the appendices.

4.8.1 Ripple-Carry Adders.

The function call to the inexact ripple-carry adder simulation, described in Section 3.2.5, is

```
[S, Cout, S0] = Adder_RCA_inexact(n, A, B, Cin, p)
```

where the input `n` is the bit length of the inputs `A` and `B`, `Cin` is the carry-in bit, and `p` is the probability of correctness of each binary computation within the adder; the output `S` is the sum S, `Cout` is the carry-out bit, and `S0` is the augmented sum S^+ defined in Eq. (25). The code for this function is given in Appendix A.1.1 on page 125. The results of this simulation are presented in Section 5.1.1.

4.8.2 Kogge-Stone Adders.

The function call to the inexact Kogge-Stone adder simulation, described in Section 3.2.7, is

```
[S, Cout, S0] = kogge_stone_inexact_PBL(n, A, B, Cin, p)
```

where the input and output parameters are the same as in Section 4.8.1. The code for this function is given in Appendix A.1.2 on page 131. The results of this simulation are presented in Section 5.1.1.

4.8.3 Ling Carry-Lookahead Adders.

The function call to the inexact Ling radix-4 carry-lookahead adder simulation, described in Section 3.2.6 is

```
[S, Cout, S0] = ling_adder_inexact_PBL(n, A, B, Cin, p)
```

where the input and output parameters are the same as in Section 4.8.1. The code for this function is given in Appendix A.1.3 on page 134. The results of this simulation are presented in Section 5.1.1.

4.8.4 Shift-and-Add Multipliers.

The function call to the inexact shift-and-add multiplier simulation, described in Section 3.3, is

```
P = Multiplier_basic_inexact(A, B, na, nb, p)
```

where `na` and `nb` are the bit lengths of the inputs `A` and `B`, and `p` is the probability of correctness of each binary operation within the multiplier; the output `P` is the product. The code for this function is given in Appendix C.3.1 on page 166. The results of this simulation are presented in Section 5.2.

4.8.5 Wallace Tree Multipliers.

The function call to the inexact Wallace tree multiplier simulation, described in Section 3.3, is

```
P = multiplier_wallace_tree_inexact_PBL(A, B, p)
```

where `A` and `B` are the inputs to the multiplier and `p` is the probability of correctness of each binary operation within the multiplier; the output `P` is the product. The code for this function is given in Appendix C.3.2 on page 170. The results of this simulation are presented in Section 5.2.

4.8.6 Floating-Point Adders.

The function call to the inexact floating-point adder simulation, described in Section 3.5.1, is

```
[Ss,Es,Ms] = Adder_floating_inexact(Sa,Ea,Ma,Sb,Eb,Mb,fmt,p)
```

where the inputs `Sa`, `Ea`, and `Ma` are the sign, exponent, and mantissa of the first addend A as shown in Eq. (64); `Sb`, `Eb`, and `Mb` are the sign, exponent, and mantissa of

the second addend B; `fmt` is a string equal to `'BINARY16'`, `'BINARY32'`, `'BINARY64'`, or `'BINARY128'` specifying the precision of the adder; and `p` is the probability of correctness of each binary operation within the adder. The outputs `Ss`, `Es`, and `Ms` are the sign, exponent, and mantissa of the sum S. The code for this function is given in Appendix B on page 151. The results of this simulation are presented in Section 5.1.3.

4.8.7 Floating-Point Multipliers.

The function call to the inexact floating-point adder simulation, described in Section 3.5.2, is

```
[Sp,Ep,Mp] = Multiplier_floating_inexact(Sa,Ea,Ma,Sb,Eb,Mb
    ,fmt,p)
```

where the inputs `Sa`, `Ea`, and `Ma` are the sign, exponent, and mantissa of the first multiplicand A as shown in Eq. (64); `Sb`, `Eb`, and `Mb` are the sign, exponent, and mantissa of the second multiplicand B; `fmt` is a string equal to `'BINARY16'`, `'BINARY32'`, `'BINARY64'`, or `'BINARY128'` specifying the precision of the multiplier; and `p` is the probability of correctness of each binary operation within the multiplier. The outputs `Ss`, `Es`, and `Ms` are the sign, exponent, and mantissa of the product P. The code for this function is given in Appendix D on page 178. The results of this simulation are presented in Section 5.3.

4.8.8 Matrix Multiplier.

Matrix multiplication is at the core of the discrete cosine transform, given in Eq. (72) and illustrated in Fig. 2. The function call to the inexact matrix multiplier simulation is

```
[C, nc] = mtimes_inexact_PBL(A, B, na, nb, p, bit)
```

where the inputs `A` and `B` are matrices, `na` and `nb` are the bit lengths of each element in `A` and `B`, and `p` is the probability of correctness of each binary operation within the multiplier. The input `bit` is the highest-order bit which can be inexact; all the lower-order bits up to and including `bit` are computed inexactly, while the higher-order bits are computed exactly. The output `C` is the matrix product, and `nc` is the bit length of each element in `C`. The code for this function is given in Appendix E on page 182.

4.8.9 Discrete Cosine Transform.

The discrete cosine transform is a key part of the JPEG compression algorithm, as shown in Fig. 2. The function call to the inexact discrete cosine transform is

```
B = DCT_inexact_PBL(A, nbits, p)
```

where the input `A` is the 8×8 matrix of image data, `nbits` is the number of bits of precision allocated to $U_{r,c}AU_{c,r}$ where U is the DCT matrix, and `p` is the probability of correctness of each binary operation performed during the DCT. For this dissertation, `nbits` is always 22. The output `B` is the discrete cosine transform of the input `A`. The code for this function is given in Appendix F.6.4.2 on page 193. The JPEG simulation results are presented in Section 5.4.2.

4.8.10 JPEG Compression Algorithm.

The inexact JPEG compression algorithm, shown in Fig. 2, is performed by the main program given in Appendix F.6.1 on page 185. This program uses the inexact DCT described in Section 4.8.9. The JPEG simulation results are presented in Section 5.4.

V. Results

We now present the results of our inexact adder, multiplier, and JPEG simulations described in Chapter IV. These simulations were performed using Matlab using a probabilistic Boolean logic (PBL) error model. These results illustrate how the likelihood and distribution of errors in the simulations vary with the probability of correctness p. This chapter also includes the results of the Spectre$^{\text{TM}}$ analog simulations which show how output error, energy consumption, and energy-delay product (EDP) vary with p.

5.1 Inexact Adders

In this section, we present the results of the Matlab PBL simulations of 8, 16, and 32-bit ripple-carry, Kogge-Stone, and Ling carry lookahead adders. These results show how the distribution of output errors varies with p. Also in this section we present the results of the Spectre$^{\text{TM}}$ simulations of 8 and 16-bit ripple-carry and Kogge-Stone adders in 14 nm FinFET CMOS technology, and 8-bit Kogge-Stone adders in 0.6 μm CMOS technology. These results show how the probability of correctness, output error, energy, and EDP vary with power supply voltage and noise power spectral density.

5.1.1 Inexact Adders with PBL.

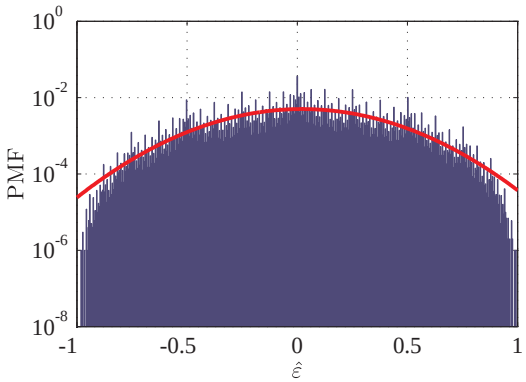
An 8-bit noisy Kogge-Stone adder was simulated using probabilistic Boolean logic, as described in Section 4.2, for various values of p. The inputs A and B were randomly drawn from a uniform distribution between 0 and $2^N - 1$; in each simulation, the sample size was 10^6. For each simulation, the noisy augmented sum $\tilde{S}^+$ (defined in (24)-(25)) and the normalized error $\hat{\varepsilon}$ (defined in (86)-(88)) were observed. Error

histograms for $p = 0.90,\ 0.95$, and 0.99 are shown in Fig. 23. The figure shows that as p increases, $\hat{\varepsilon}$ is more tightly dispersed around 0. Each histogram has a primary mode at $\hat{\varepsilon} = 0$ and multiple other modes at powers of $\frac{1}{2}$. The modes are at powers of $\frac{1}{2}$ because the most frequent errors involve only a single binary digit within the adder. For $p \geq 0.99$ the distribution of $\hat{\varepsilon}$ is highly kurtotic—so much so that a semilogarithmic scale is necessary to display the tails of the distribution. In this case, a high-order normal product distribution is the best fit to the sample, which is characterized by an extremely tall, narrow peak at $\hat{\varepsilon} = 0$, and extremely infrequent, but widely dispersed, nonzero values of $\hat{\varepsilon}$.

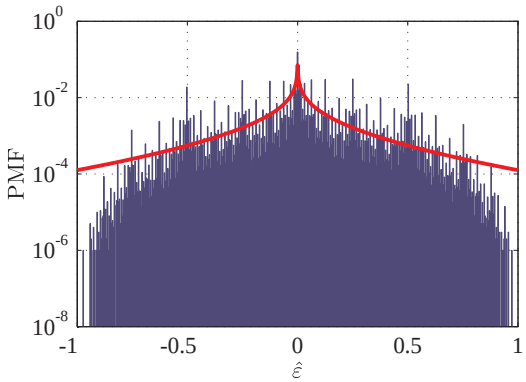
Error statistics for various values of p are summarized in Table 8. The table shows that as p approaches 1, the error standard deviation approaches 0 and the kurtosis increases. An error spectrum plot for $p = 0.90$ is shown in Fig 24. This plot shows the sample mean of 100 observations of $\hat{\varepsilon}$ at every possible value of A and B. The expected error appears roughly to be a linear function of $A + B$, with the smallest error magnitude along the line $B = 255 - A$. Closer inspection reveals discontinuities along the lines $A = 64$, $A = 128$, $A = 192$, $B = 64$, $B = 128$, and $B = 192$.

5.1.2 Probability Distributions.

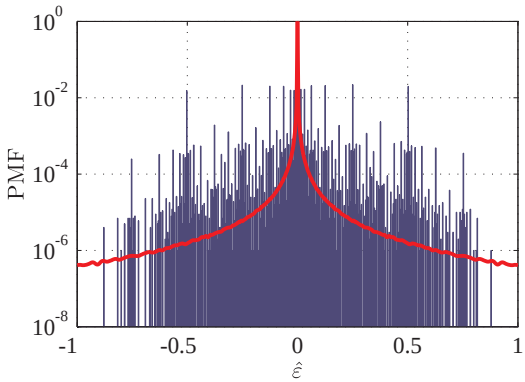
For each 2, 4, and 8-bit inexact adder, a maximum likelihood estimation was performed on the distribution of the normalized error $\hat{\varepsilon}$. MLE was also performed on a 6-bit ripple-carry adder, and a 16-bit Ling carry lookahead adder. Three different types of distributions were considered: Gaussian, Laplacian, and normal product distributions with order $2 \leq \psi \leq 40$. The results for the ripple-carry adder are shown in Fig. 26. The Gaussian distribution seemed to be the best fit for values of $p \leq 0.85$, and for higher values of p, a normal product distribution was usually the best fit. The exception was the 2-bit ripple-carry adder, for which the Laplacian distribution was



(a) $p = 0.90$. Gaussian fit with $\tilde{\sigma} = 0.31$.



(b) $p = 0.95$. NP3 fit with $\tilde{\sigma} = 0.37$.



(c) $p = 0.99$. NP20 fit with $\tilde{\sigma} = 0.13$.

Figure 23. Error histograms showing the probability mass function (PMF) of $\hat{\varepsilon}$ for an inexact 8-bit Kogge-Stone adder, with inputs A and B uniformly distributed from 0 to $2^N - 1$, for various values of p. Results are from Matlab PBL simulation. The red line in (a) is a Gaussian fit to the PMF, and in (b)-(c) the red lines are normal product (NP) curve fits based on maximum likelihood estimation. The slight asymmetry in the figures is due to the fact that the data are from Monte Carlo simulations based on random number generation.

Table 8. Error Statistics: 8-Bit Kogge-Stone Adder with PBL

p	$P(\hat{\varepsilon} = 0)$	Mean	Std Dev	Skewness	Kurtosis
0.8000	0.0056	0.0207	0.3468	-0.0424	2.46
0.8500	0.0109	0.0233	0.3366	-0.0406	2.54
0.9000	0.0319	0.0217	0.3125	-0.0193	2.78
0.9500	0.1473	0.0143	0.2573	0.0402	3.65
0.9600	0.2116	0.0125	0.2382	0.0716	4.12
0.9700	0.3061	0.0098	0.2130	0.1061	4.95
0.9800	0.4484	0.0068	0.1805	0.1634	6.59
0.9900	0.6668	0.0036	0.1323	0.2809	11.6
0.9950	0.8150	0.0018	0.0957	0.4194	21.6
0.9960	0.8493	0.0013	0.0857	0.4571	26.8
0.9970	0.8843	0.0011	0.0748	0.5682	34.9
0.9980	0.9209	0.0008	0.0609	0.7184	52.2
0.9990	0.9596	0.0004	0.0434	1.0618	102
0.9995	0.9797	0.0002	0.0310	1.5608	202
0.9996	0.9837	0.0002	0.0277	2.0177	250
0.9997	0.9876	0.0001	0.0243	1.9396	328
0.9998	0.9919	0.0001	0.0196	1.8764	496
0.9999	0.9959	0.0001	0.0138	3.9838	1007

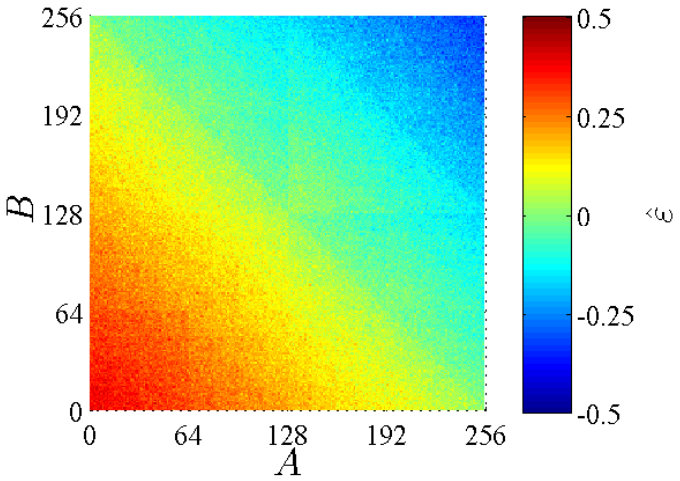
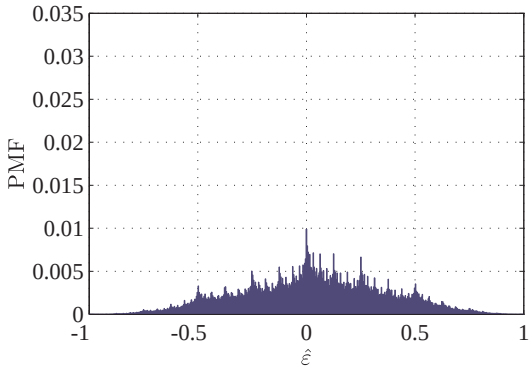
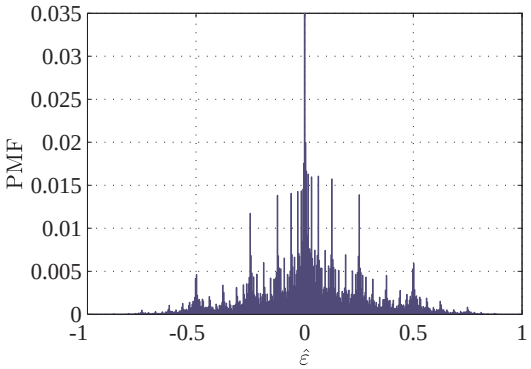


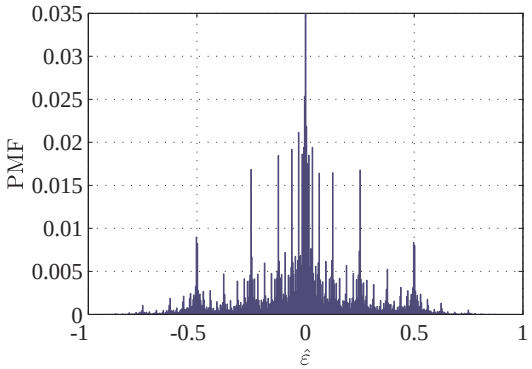
Figure 24. Error spectrum for an inexact 8-bit Kogge-Stone adder, for all possible values of the inputs A and B, and probability $p = 0.90$ of correctness at each node. Each point on the plot is the sample mean of 100 observations of $\hat{\varepsilon}$.



(a) Kogge-Stone adder. Histogram peak value is 0.0099.



(b) Ripple-Carry adder. Histogram peak value is 0.0457.



(c) Ling adder. Histogram peak value is 0.0810.

Figure 25. Error histograms for various inexact 32-bit adders, with $p = 0.90$, and with inputs A and B uniformly distributed from 0 to $2^N - 1$. Results are from Matlab PBL simulation. The modes are at powers of $\frac{1}{2}$ because the most frequent errors involve only a single binary digit within the adder.

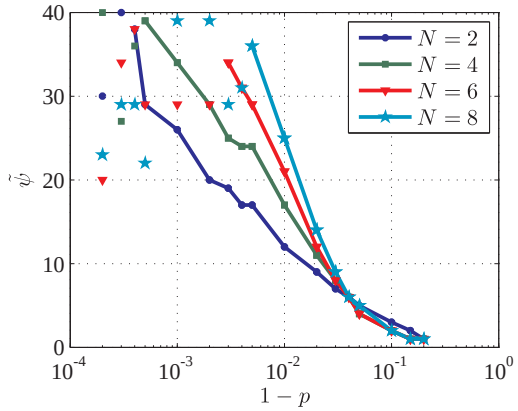
usually the most likely fit. From Fig. 26a it is apparent that $\tilde{\psi}$ decreases monotonically with $(1-p)$, for smaller values of p. The other adder architectures displayed this relationship as well. In each case, $\tilde{\psi}$ seems to increase as p increases, until $\tilde{\psi}$ reaches 40, after which its behavior becomes erratic. This erratic behavior may be due to the fact that we only attempted to fit NP distributions with $\tilde{\psi} \leq 40$, and may indicate that the fits obtained were suboptimal. The monotonic relationship is indicated by the bold lines connecting the points in Fig. 26a, and also the associated values of $\tilde{\sigma}$ in Fig. 26b.

Fig. 26c shows the estimated order $\tilde{\psi}$ as a function of the adder bit width N. Looking again at only the non-erratic points mentioned above, Fig. 26c indicates that for $N \leq 8$, $\tilde{\psi}$ is roughly a linear function of N, with slope that increases with p.

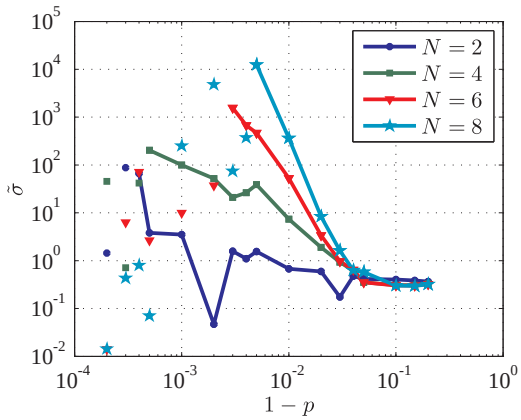
Fig. 26d illustrates the sample kurtosis of $\hat{\varepsilon}$ as a function of $(1-p)$. It is apparent that the kurtosis is almost independent of N. Also, the log of the kurtosis is nearly a linear function of $\log(1-p)$. As p increases, so does kurtosis, which is also associated with an increase in the order $\tilde{\psi}$, as we would expect in accordance with Fig. 26a.

5.1.3 Comparisons Among Adder Types.

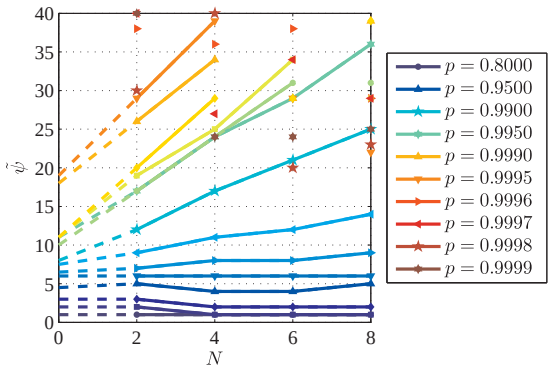
Figs. 25 and 27 compare the performances of the ripple-carry, Kogge-Stone and Ling carry lookahead architectures. Fig. 25 shows histograms for each type of 32-bit adder for a value of $p = 0.90$. It is apparent that each histogram has a primary mode at $\hat{\varepsilon} = 0$ and other modes at powers of $\frac{1}{2}$. Fig. 27 provides summary statistics for 8, 16, and 32-bit ripple-carry, Kogge-Stone, and Ling adders for various values of p between 0.8000 and 0.9999. Fig. 27a shows that the probability of zero error $(\hat{\varepsilon} = 0)$ increases with p and decreases with N. Fig. 27b shows that the standard deviation of $\hat{\varepsilon}$ decreases with p, and is smaller for the ripple-carry and Ling adders than for the Kogge-Stone adder. Interestingly, the 32-bit version of the Ling adder



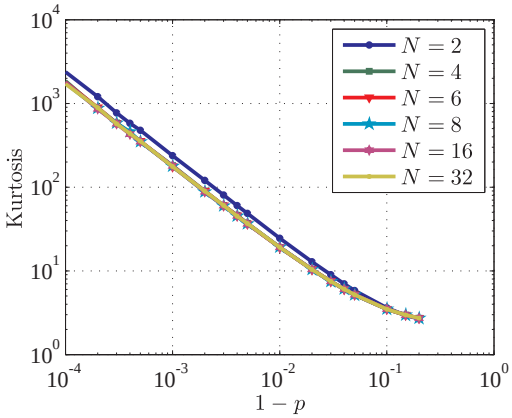
(a) Normal product order $\tilde{\psi}$ as a function of $(1 - p)$.



(b) Normal product scale parameter $\tilde{\sigma}$ as a function of $(1 - p)$.



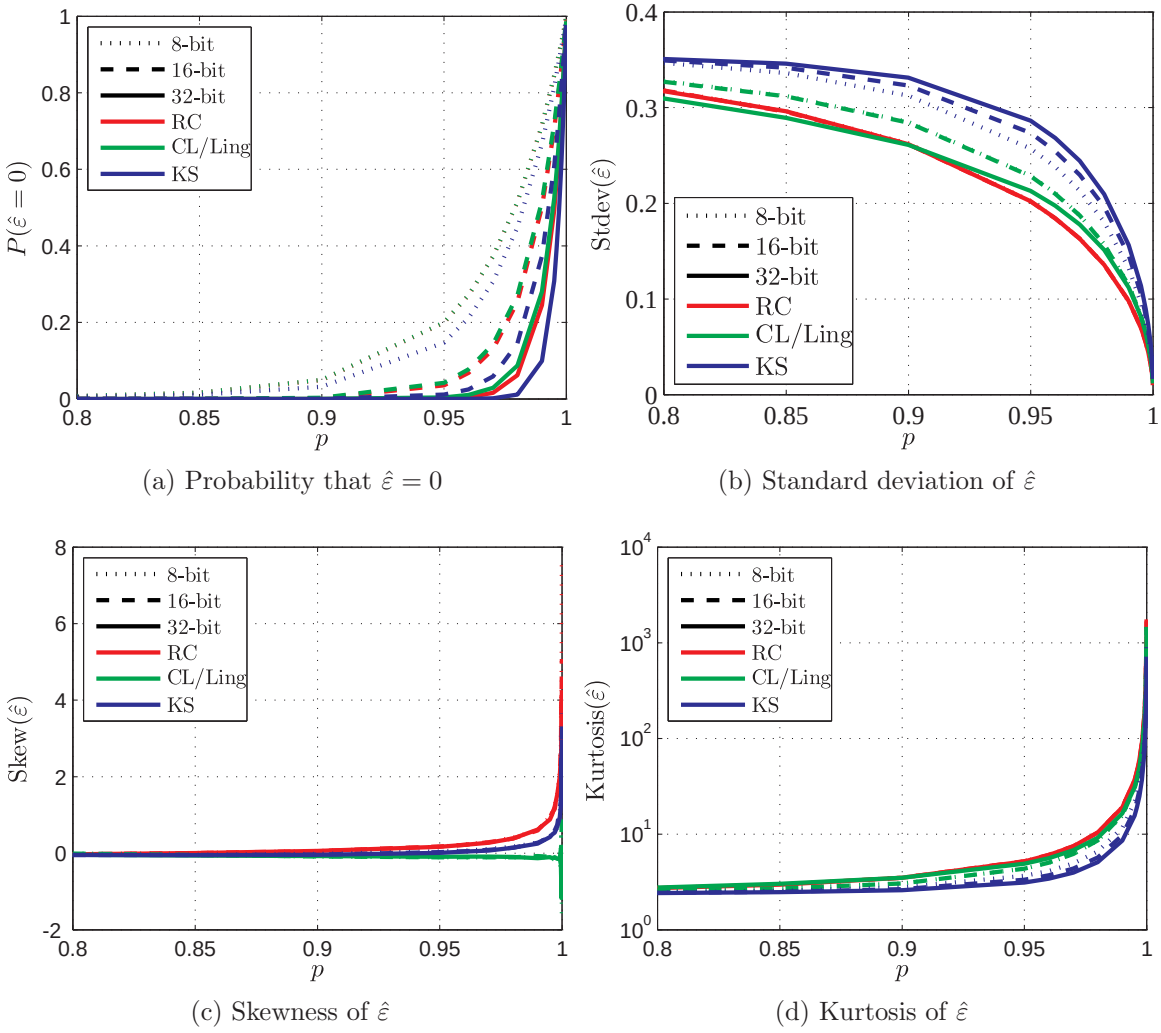
(c) Normal product order $\tilde{\psi}$ as a function of N.



(d) Sample kurtosis as a function of $(1 - p)$.

Figure 26. Error statistics for inexact N-bit ripple-carry adders with various values of N and p, and with inputs A and B uniformly distributed from 0 to $2^N - 1$. Results are from Matlab PBL simulation. (a,b,c) Most likely normal product distribution to fit each sample of $\hat{\varepsilon}$. Due to resource constraints, distribution fits with $\tilde{\psi} > 40$ were not attempted, resulting in some suboptimal values of $\tilde{\psi}$; these are the points *not* joined by lines. (d) Sample kurtosis of $\hat{\varepsilon}$.

has a smaller dispersion of $\hat{\varepsilon}$ than the 8 and 16-bit versions. Fig. 27c shows very very little skewness, except when p is large. Fig. 27 shows that the distribution of $\hat{\varepsilon}$ is exceedingly kurtotic for large values of p.



(a) Probability that $\hat{\varepsilon} = 0$

(b) Standard deviation of $\hat{\varepsilon}$

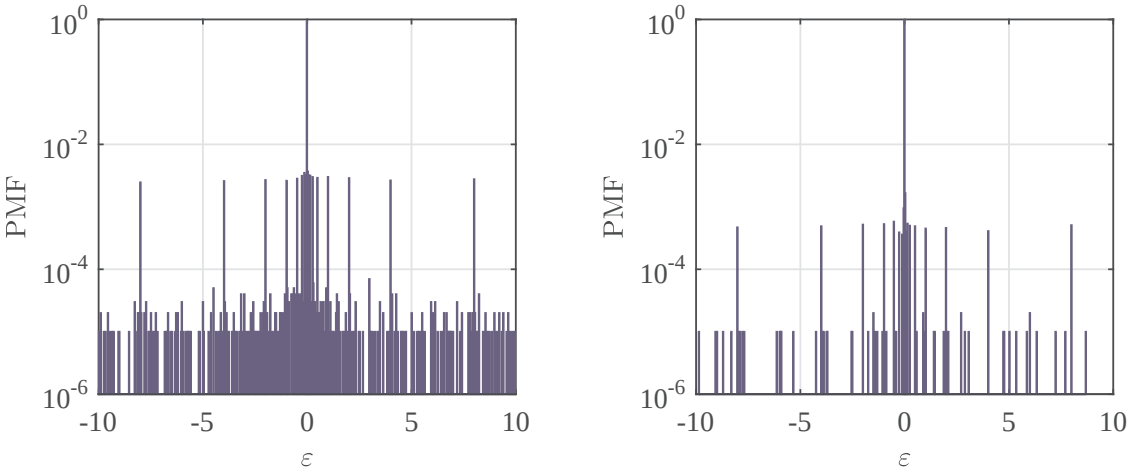
(c) Skewness of $\hat{\varepsilon}$

(d) Kurtosis of $\hat{\varepsilon}$

Figure 27. Error statistics for various inexact adders, with inputs A and B uniformly distributed from 0 to $2^N - 1$. Results are from Matlab PBL simulation.

We considered using IEEE 754 standard floating-point adders and multipliers for use in the JPEG compression algorithm. Initial results were not promising for this purpose, due to large errors occurring frequently, and we were able to implement the JPEG algorithm using integer arithmetic. The results are included here for completeness.

Error histograms for various floating-point adders are shown in Fig. 28. The modes of the distribution are at powers of 2.



(a) Half-precision adder with $p = 0.9999$.

(b) Single-precision adder with $p = 0.99999$.

Figure 28. Error histograms for inexact 16 and 32-bit floating-point adders, with inputs A and B uniformly distributed from -100 to $+100$, for various values of p. Results are from Matlab PBL simulation.

5.1.4 Spectre™ Simulation.

The Spectre™ simulation of the noisy 0.6 μm Kogge-Stone adder, described in Section 4.1.1, produced the results shown in Table 9. These results show that increased energy consumption is necessary to reduce error, and the relationship is nonlinear. For example, when the noise PSD is 5×10^{-10} V^2/Hz, $V_{DD} = 1.5$ V, $E_{avg} = 2.32$ pJ per cycle, and the normalized error standard deviation is 0.1827; when $V_{DD} = 3.3$ V, E_{avg} increases to 12.70 pJ, and the error standard deviation decreases to 0.0584. As expected, increasing energy consumption also reduces delay. For example, Table 9 shows that under noise-free conditions, when $V_{DD} = 1.5$ V the energy consumption is 1.41 pJ per cycle and the delay is 12.28 ns, and when $V_{DD} = 3.3$ V the energy consumption increases to 9.55 pJ and the delay decreases to 2.89 ns. An error histogram is shown in Fig. 29. The errors in Table 9 are smaller than the Matlab PBL simula-

tion errors shown in Table 8. The Spectre^{TM} simulation errors in Fig. 29 also appear to be smaller than the PBL errors in Fig. 23a. The reason the binary error model produces greater output errors than the analog model is that with the binary model, the error at each is either 0% or 100%, whereas the analog model allows a continuum of errors at each node. With the analog model, an error voltage may slightly exceed the $V_{DD}/2$ threshold for a brief amount of time, but may not have sufficient energy to propagate throughout the circuit.

Plots of energy savings (i.e. percent reduction in E_{avg}) and Energy-Delay Product (EDP) savings as functions of the error $\hat{\varepsilon}$ are shown in Fig. 32-33. Fig. 32 shows that energy savings drop sharply as $\hat{\varepsilon}$ approaches zero. Scaling the supply voltage increases delay. In spite of this trade-off, Fig. 33 shows that there is still a benefit in terms of EDP.

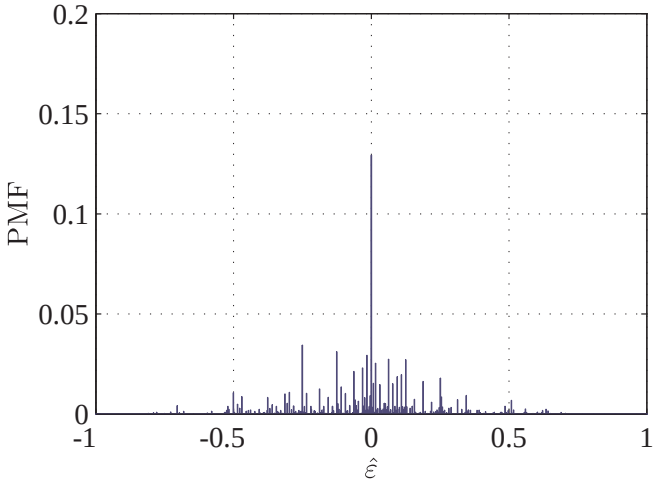


Figure 29. Error histogram for an 8-bit Kogge-Stone adder, with inputs A and B uniformly distributed from 0 to $2^N - 1$. Power supply $V_{DD} = 1.5$ V; noise PSD is 1×10^{-9} V^2/Hz. Result is from a Spectre^{TM} simulation of 0.6 μm CMOS technology.

Fig. 30 shows one main result of this research. This figure shows the energy reduction (percentage) as a function of the percentage error rate $(1 - p)$ for two adder architectures. Results are shown for six values of the noise power spectral density (a)-(f). The figure shows the energy reduction as a function of error rate for an 8-bit

Table 9. Error Statistics: Noisy 8-Bit Kogge-Stone Adder Spectre™ Simulation

V_{DD} [V]	δ_{max} [ns]	Noise-Free E_{avg} [pJ]
1.5	12.28	1.41
2.0	6.39	3.49
2.5	4.73	5.73
3.0	3.32	8.27
3.3	2.89	9.55

Noise #1: 5×10^{-10} V^2/Hz

V_{DD} [V]	E_{avg} [pJ]	Mean	Std Dev	Skewness	Kurtosis
1.5	2.32	0.0080	0.1827	-0.0183	6.09
2.0	4.42	-0.0031	0.1085	-1.0517	22.5
2.5	7.03	-0.0011	0.0536	-0.4252	44.4
3.0	10.42	-0.0019	0.0582	-0.2743	48.3
3.3	12.70	-0.0013	0.0584	-0.2544	50.9

Noise #2: 1×10^{-9} V^2/Hz

V_{DD} [V]	E_{avg} [pJ]	Mean	Std Dev	Skewness	Kurtosis
1.5	3.53	-0.0219	0.2327	-0.1222	3.93
2.0	7.09	-0.0270	0.1746	-0.7496	7.33
2.5	11.51	-0.0151	0.1319	-1.0512	11.1
3.0	16.46	-0.0150	0.1056	-1.8386	17.2
3.3	19.46	-0.0098	0.0945	-2.2123	22.7

ripple carry adder and a 16-bit ripple carry adder, both in 14 nm FinFET CMOS technology. The figure also shows the energy reduction as a function of error rate for an 8-bit Kogge Stone adder in a 16-bit Kogge Stone adder, both in 14 nm FinFET CMOS technology. The figure also shows the energy reduction for an 8-bit Kogge Stone adder in a 0.6 μm CMOS technology.

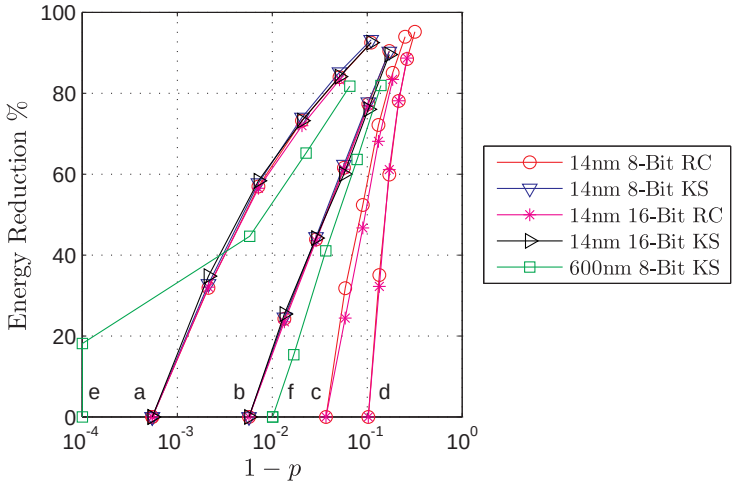


Figure 30. Percent reduction in energy E_{avg} per switching cycle, as a function of $1-p$, for the noisy 8 and 16-bit ripple-carry (RC) and Kogge-Stone (KS) adders in 0.6 μm CMOS and 14 nm FinFET CMOS technologies. $1-p$ is the probability of error at each node within the adder circuit. Curves (a) through (f) represent the noise conditions specified by their power spectral densities: (a) 3×10^{-12} V^2/Hz, (b) 5×10^{-12} V^2/Hz, (c) 1×10^{-11} V^2/Hz, (d) 2×10^{-11} V^2/Hz, (e) 5×10^{-10} V^2/Hz, and (f) 1×10^{-9} V^2/Hz. Results are from Spectre^{TM} simulation.

The results for these adders in Fig. 30 show that the error reduction tends to increase as the error rate increases, for each technology. The results show that the error reduction takes on the largest value of over 90% when the error rate exceeds 0.1. For example, for the case in which the noise power spectral density takes on a value of 3×10^{-12} V^2/Hz (see 'a' curves in the figure), the results in the figure show that an energy reduction of approximately 92% can be achieved with an error rate of 0.1 for the 8-bit ripple carry adder, 16-bit ripple carry adder, 8-bit KS adder, and 16-bit KS adder. The results in the figure show that, at each value of $(1-p > 0.005)$, the energy reduction achieved with an 8-bit KS adder in 14 nm FinFET CMOS technology with a

noise power spectral density of 3×10^{-12} V^2/Hz is greater than the energy reduction achieved with an 8-bit KS adder in 0.6 μm CMOS technology with a noise power spectral density of 5×10^{-12} V^2/Hz.

Fig. 31 shows the reduction in the energy-delay product per switching cycle as a function of $1-p$, for 8-bit ripple carry adder, a 16-bit ripple carry adder, an 8-bit KS adder, a 16-bit KS adder, and a 0.6 μm 8-bit KS adder. For the lowest values of the noise power spectral density, the results in the figure show that the reduction in the energy-delay product can take on a maximum value for a certain choice of $(1-p)$. For example for the lowest value of the noise power spectral density of 3×10^{12} V^2/Hz (curves labeled 'a' curves), the energy-delay product achieves a maximum reduction for the 8-bit KS adder in 14 nm FinFET CMOS technology (blue curve with triangles) when the value of $(1-p)$ is approximately 0.05. The results in the figure show that the reduction in the energy-delay product tends to be smaller as the noise power spectral density is increased.

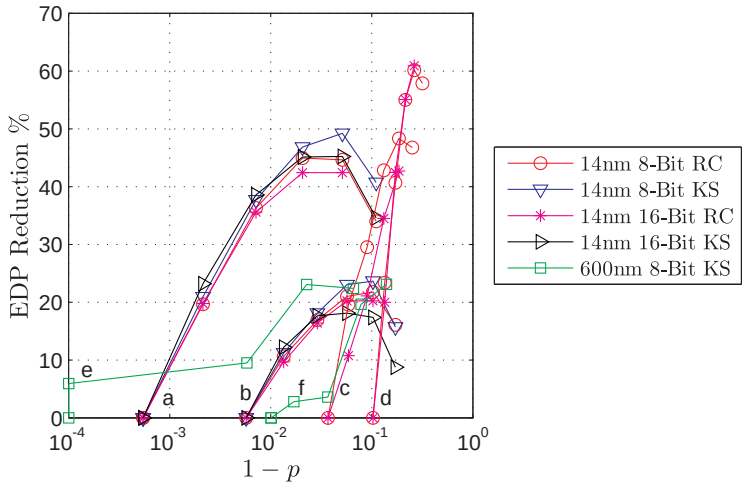


Figure 31. Percent reduction in energy-delay product (EDP) per switching cycle, as a function of $1-p$, for the noisy 8 and 16-bit ripple-carry (RC) and Kogge-Stone (KS) adders in 0.6 μm CMOS and 14 nm FinFET CMOS technologies. $1-p$ **is the probability of error at each node within the adder circuit. Curves (a) through (f) represent the noise conditions specified by their power spectral densities: (a)** 3×10^{-12} **V^2/Hz, (b)** 5×10^{-12} **V^2/Hz, (c)** 1×10^{-11} **V^2/Hz, (d)** 2×10^{-11} **V^2/Hz, (e)** 5×10^{-10} **V^2/Hz, and (f)** 1×10^{-9} **V^2/Hz. Results are from Spectre^{TM} simulation.**

Fig. 32 shows another main result of this dissertation. This figure shows the energy reduction as a function of the standard deviation of the normalized output error for an 8-bit ripple carry adder and a 16-bit ripple carry adder in 14 nm FinFET CMOS technology. The figure also shows the energy reduction as a function of the standard deviation of the normalized output error for an 8-bit KS adder and a 16-bit KS adder in 14 nm FinFET CMOS technology, and for an 8-bit KS adder in 0.6 μm CMOS technology. Results are shown for six values of the noise power spectral density, (a)-(f).

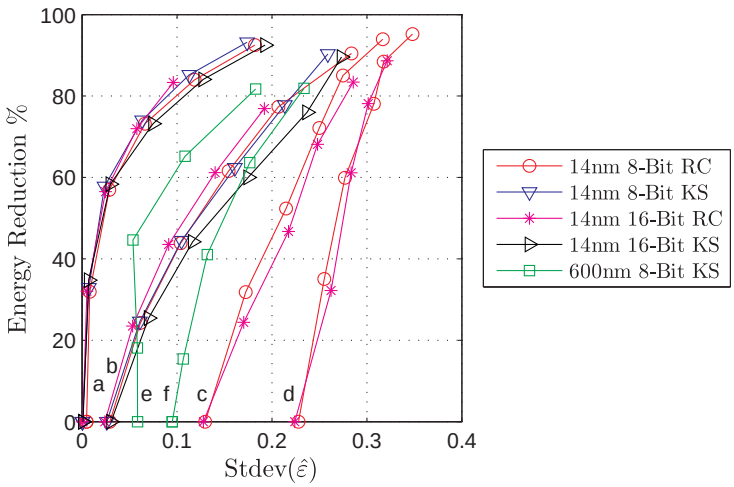


Figure 32. Percent reduction in energy E_{avg} per switching cycle, as a function of the standard deviation of the normalized output error $\hat{\varepsilon}$, for the noisy 8 and 16-bit ripple-carry (RC) and Kogge-Stone (KS) adders in 0.6 μm CMOS and 14 nm FinFET CMOS technologies. Curves (a) through (f) represent the noise conditions specified by their power spectral densities: (a) 3×10^{-12} V^2/Hz, (b) 5×10^{-12} V^2/Hz, (c) 1×10^{-11} V^2/Hz, (d) 2×10^{-11} V^2/Hz, (e) 5×10^{-10} V^2/Hz, and (f) 1×10^{-9} V^2/Hz. Results are from Spectre^{TM} simulation.

The results for these adders in Fig. 32 show that the error reduction in each adder architecture tends to take on the largest value, exceeding 90%, when the standard deviation of the error exceeds approximately 0.18. For example, for the case in which the noise power spectral density takes on a value of 3×10^{-12} V^2/Hz (see 'a' curves in the figure), the results in the figure show that the energy reduction of approximately 95% can be achieved with a standard deviation of the normalized output error of 0.18

for the 8-bit ripple carry adder, 16-bit ripple carry adder, 8-bit KS adder, and 16-bit KS adder.

Fig. 33 shows the reduction in the energy-delay product per switching cycle as a function of the standard deviation of the normalized output error. The results in this figure show that the 8-bit ripple-carry adder and 16-bit ripple carry adder in 14 nm FinFET CMOS technology achieve the greatest reduction in the energy-delay product of 60% when the noise power spectral density takes on the value of 2×10^{-11} V^2/Hz, as shown in the 'd' curves, where the standard deviation of the error is approximately 0.32. The results for the 8-bit KS adder and 16-bit KS adder show that as the noise power spectral density is increased, the energy-delay product benefit is reduced from 50% to 22% (see 'a' curves compared with 'b' curves).

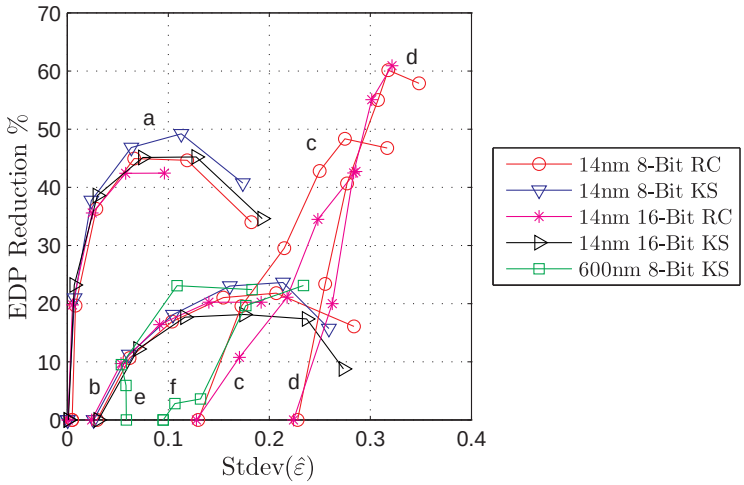


Figure 33. Percent reduction in energy-delay product (EDP) per switching cycle, as a function of the standard deviation of the normalized output error $\hat{\varepsilon}$, for the noisy 8 and 16-bit ripple-carry (RC) and Kogge-Stone (KS) adders in 0.6 μm CMOS and 14 nm FinFET CMOS technologies. Curves (a) through (f) represent the noise conditions specified by their power spectral densities: (a) 3×10^{-12} V^2/Hz, (b) 5×10^{-12} V^2/Hz, (c) 1×10^{-11} V^2/Hz, (d) 2×10^{-11} V^2/Hz, (e) 5×10^{-10} V^2/Hz, and (f) 1×10^{-9} V^2/Hz. Results are from Spectre$^{\text{TM}}$ simulation.

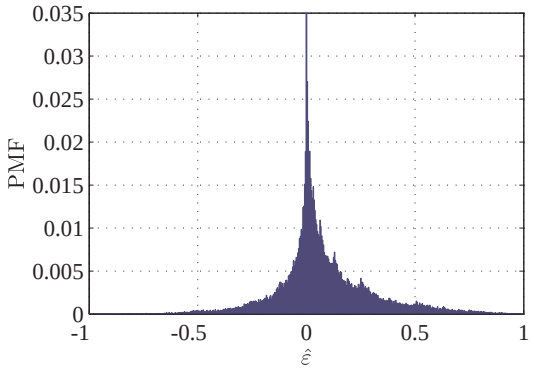
Plots of energy consumption E_{avg} and energy-delay product (EDP) as functions of the error $\hat{\varepsilon}$ are shown in Fig. 32. This figure shows that E_{avg} greatly increases as $\hat{\varepsilon}$ approaches zero. Scaling the supply voltage increases delay. In spite of this

trade-off, Fig. 33 shows that there is still a benefit in terms of EDP, as mentioned in the previous paragraph.

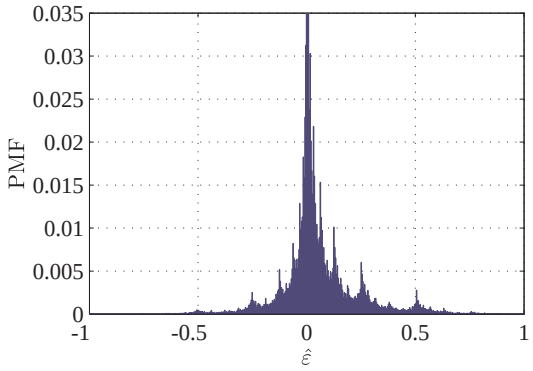
A similar result for the 14 nm ripple-carry adders is shown in Figs. 32-33. Fig. 32 confirms that, in a noisy circuit, energy savings can be achieved by allowing more errors at the output. Fig. 33 shows that EDP improvements can be made by allowing more errors, up to a point. However, if the error standard deviation is allowed to increase beyond 0.12 (corresponding to $V_{dd} < 0.4$ V in curve (a)), the increased delay begins to dominate the EDP. Therefore, correctness can only be traded for EDP up to that point. A designer who is interested only in saving energy, and not concerned about speed, would use Figs. 30 and 32 to choose acceptable values of p, output error, and energy reduction. However, a designer concerned with delay could instead use the EDP curves in Figs. 31 and 33.

5.2 Shift-and-Add Multiplier with PBL

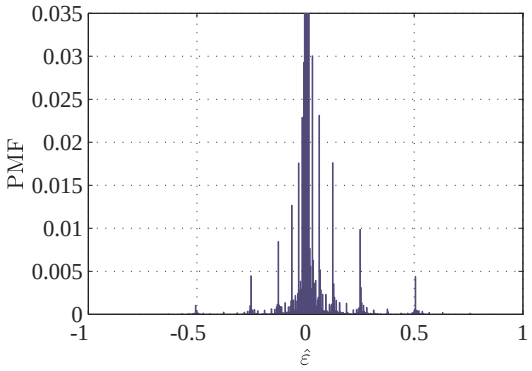
A 16-bit noisy shift-and-add multiplier was simulated using probabilistic Boolean logic, as described in Section 4.4, for various values of p. The inputs A and B were randomly drawn from a uniform distribution between 0 and $2^N - 1$; in each simulation, the sample size was 10^5. For each simulation, the noisy product $\tilde{P}$ and the normalized error $\hat{\varepsilon}$ were observed. Error histograms for $p = 0.90,\ 0.95$, and 0.99 are shown in Fig. 34. The figure shows that as p increases, $\hat{\varepsilon}$ is more tightly dispersed around 0. Each histogram has a primary mode at $\hat{\varepsilon} = 0$ and multiple other modes at powers of $\frac{1}{2}$. Error statistics for various values of p are summarized in Table 10. Future work will include Spectre$^{\text{TM}}$ analog simulations of 16-bit shift-and-add multipliers.



(a) $p = 0.90$. Histogram peak value is 0.035.

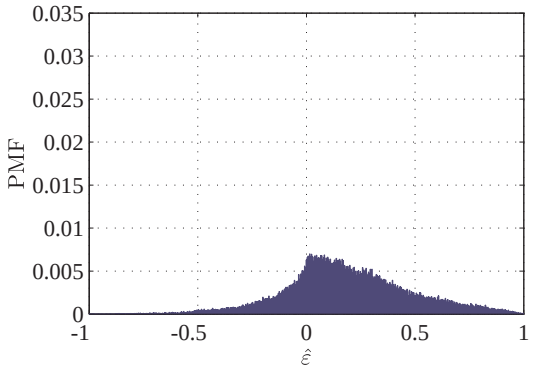


(b) $p = 0.95$. Histogram peak value is 0.078.

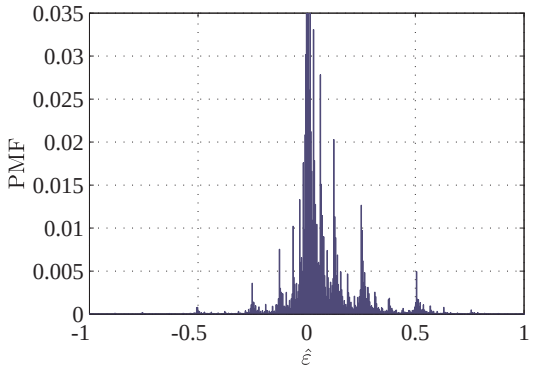


(c) $p = 0.99$. Histogram peak value is 0.456.

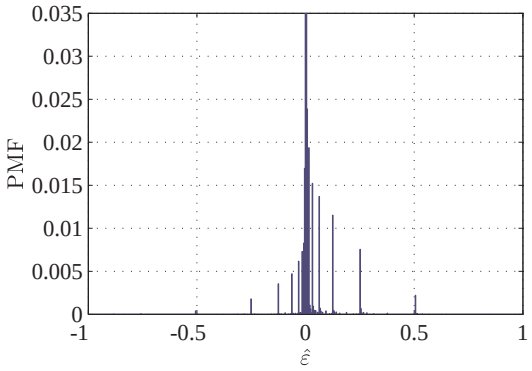
Figure 34. Error histograms for an inexact 16-bit shift-and-add multiplier, with inputs A and B uniformly distributed from 0 to $2^N - 1$, for various values of p. Results are from Matlab PBL simulation.



(a) $p = 0.95$. Histogram peak value is 0.007.

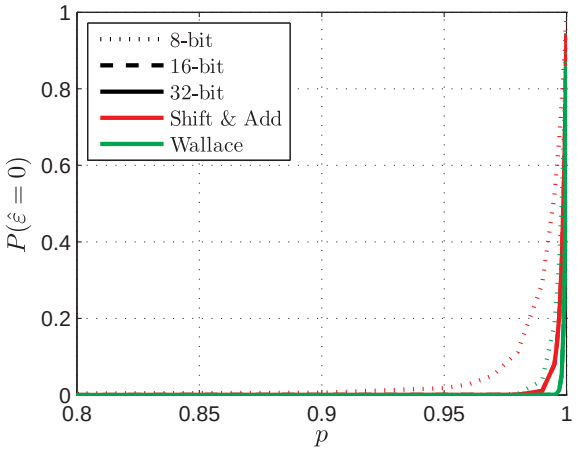


(b) $p = 0.99$. Histogram peak value is 0.135.

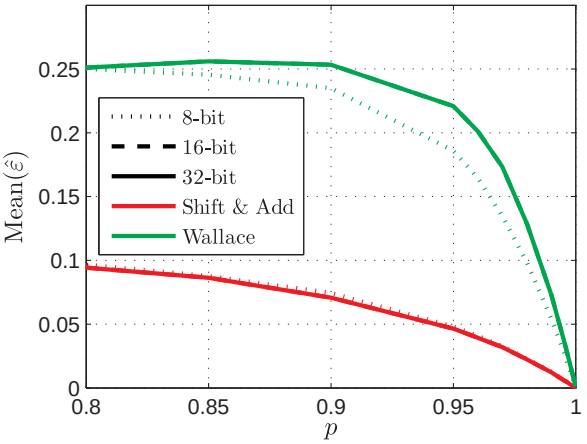


(c) $p = 0.999$. Histogram peak value is 0.799.

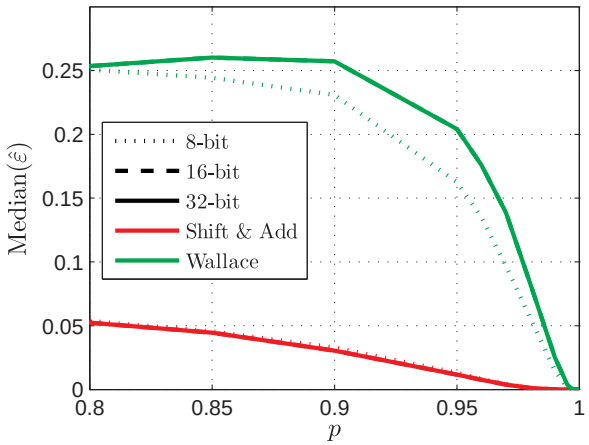
Figure 35. Error histograms for an inexact 16-bit Wallace tree multiplier, with inputs A and B uniformly distributed from 0 to $2^N - 1$, for various values of p. Results are from Matlab PBL simulation.



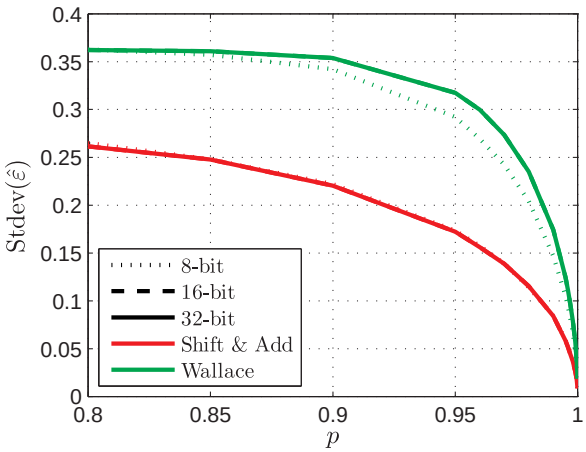
(a) Probability that $\hat{\varepsilon} = 0$



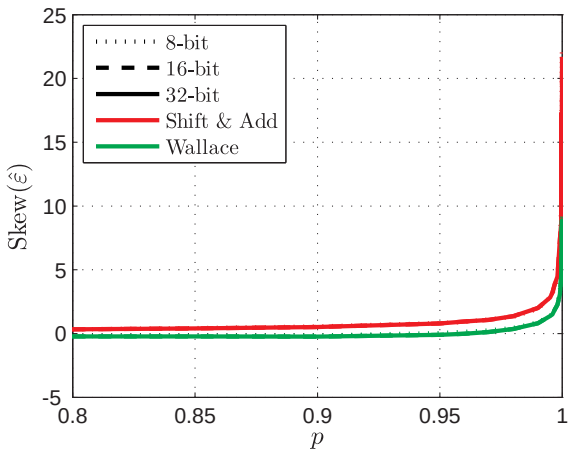
(b) Mean of $\hat{\varepsilon}$



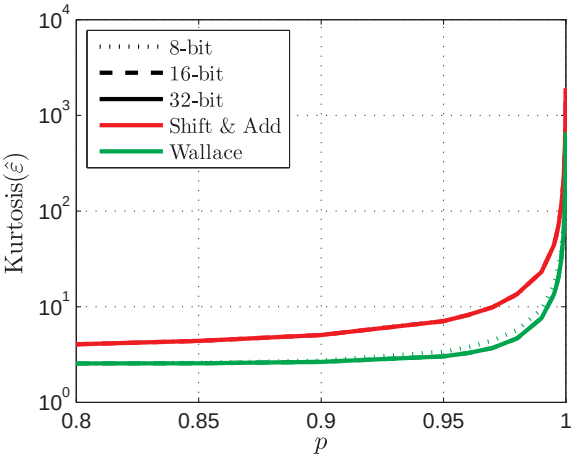
(c) Median of $\hat{\varepsilon}$



(d) Standard deviation of $\hat{\varepsilon}$



(e) Skewness of $\hat{\varepsilon}$



(f) Kurtosis of $\hat{\varepsilon}$

Figure 36. Error statistics for various inexact multipliers, with inputs A and B uniformly distributed from 0 to $2^N - 1$. Results are from Matlab PBL simulation.

Table 10. Error Statistics: 16-bit Shift-and-Add Multiplier with PBL

p	$P(\hat{\varepsilon} = 0)$	Mean	Std Dev	Skewness	Kurtosis
0.8000	0.0042	0.0966	0.2650	0.3295	4.04
0.8500	0.0047	0.0878	0.2478	0.4303	4.37
0.9000	0.0056	0.0745	0.2221	0.5528	5.07
0.9500	0.0177	0.0481	0.1733	0.8228	7.14
0.9600	0.0277	0.0408	0.1576	0.9201	8.09
0.9700	0.0513	0.0327	0.1389	1.1144	9.91
0.9800	0.1124	0.0230	0.1140	1.3123	13.2
0.9900	0.2901	0.0126	0.0842	1.9383	23.6
0.9950	0.5215	0.0064	0.0591	2.8432	42.4
0.9960	0.5915	0.0052	0.0529	3.0746	53.9
0.9970	0.6685	0.0040	0.0462	3.7838	71.1
0.9980	0.7645	0.0025	0.0374	4.1237	105
0.9990	0.8731	0.0012	0.0266	5.7969	203
0.9995	0.9330	0.0007	0.0186	10.5104	419
0.9996	0.9469	0.0004	0.0160	9.7384	537
0.9997	0.9591	0.0004	0.0146	16.2839	713
0.9998	0.9719	0.0004	0.0127	22.1914	928
0.9999	0.9859	0.0001	0.0082	13.4407	1965

5.3 Comparisons Among Multiplier Types

Fig. 36 provides summary statistics for 16, 32, and 64-bit shift-and-add multipliers for various values of p between 0.8000 and 0.9999. Fig. 36a shows that the probability of zero error ($\hat{\varepsilon} = 0$) increases with p and decreases with N. Figs. 36b-c show that, for the Wallace tree with $p < 0.98$, the mean and median of $\hat{\varepsilon}$ are much larger than for the shift and add multiplier. For example, for a 64-bit Wallace tree multiplier with $p = 0.95$, the mean of $\hat{\varepsilon}$ is 0.2208 and the median is 0.2042, while for the 64-bit shift-and-add multiplier the mean error is 0.0464 and the median is 0.0117. This nonzero mean is also evident in Fig. 35a. Fig. 36d shows that the standard deviation of $\hat{\varepsilon}$ decreases with p. For example, for the 64-bit Wallace tree multiplier, when $p = 0.95$ the standard deviation of $\hat{\varepsilon}$ is 0.3174; when $p = 0.99$ the error standard deviation drops to 0.1743. From Fig. 36d, we can see that the dispersion of error is greater for the Wallace tree than for the shift and add multiplier. For example, when

$p = 0.90$, the error standard deviation is 0.3420 for the 16-bit Wallace tree multiplier and 0.3538 for the 32 and 64-bit Wallace tree multipliers, but only 0.2204 for the 16, 32, and 64-bit shift-and-add multipliers. Fig. 36e shows slight skewness for small p, which greatly increases when p is large. Fig. 36f shows that the distribution of $\hat{\varepsilon}$ is exceedingly kurtotic for large values of p. For example, when $p \geq 0.998$, the kurtosis of $\hat{\varepsilon}$ for the 16, 32, and 64-bit shift-and-add multipliers is greater than 100. Figs. 36b-f show that the bit width of the multiplier has very little effect on the mean, median, standard deviation, skewness, and kurtosis of the error distribution.

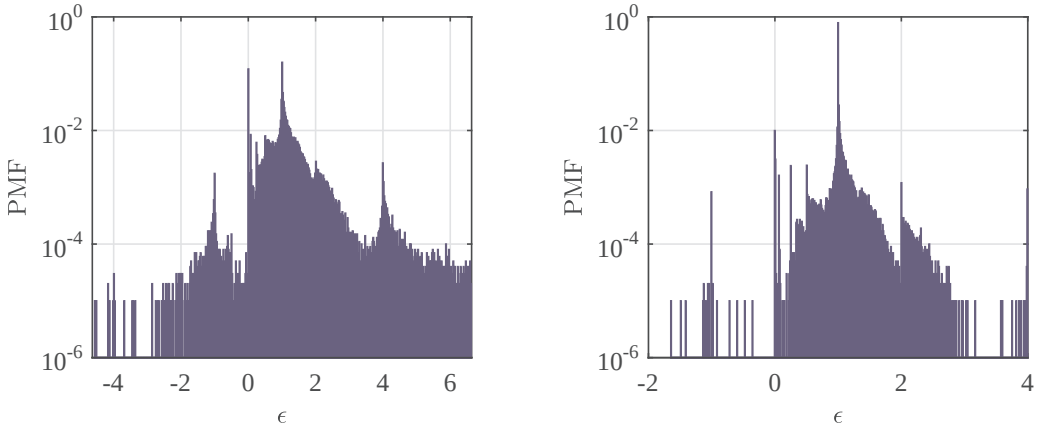
We considered using IEEE 754 standard floating-point adders and multipliers for use in the JPEG compression algorithm. As was mentioned in Section 5.1.3, the initial results were not promising for this purpose, due to large errors occurring frequently, and we were able to implement the JPEG algorithm using integer arithmetic. The results are included here for completeness. When viewing multiplier performance in terms of the product error metric ϵ, floating-point multiplier overall errors vary tremendously with p, as shown in Figs. 37-38.

5.4 JPEG Image Compression

The JPEG compression algorithm, repeated in Fig. 39, brings together the methodologies described througout this dissertation. The color space transform, discrete cosine transform, and quantization can all be performed using inexact adders and multipliers. In this work, we show the results of the inexact color space transformation and inexact DCT on the performance of the JPEG algorithm.

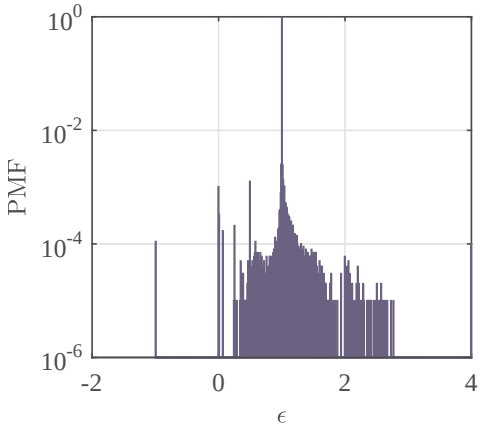
5.4.1 Inexact Color Space Transform.

This section illustrates the result of performing an inexact CST described in Section 4.6 and shown in Fig. 39. Performance results for the inexact CST are shown in



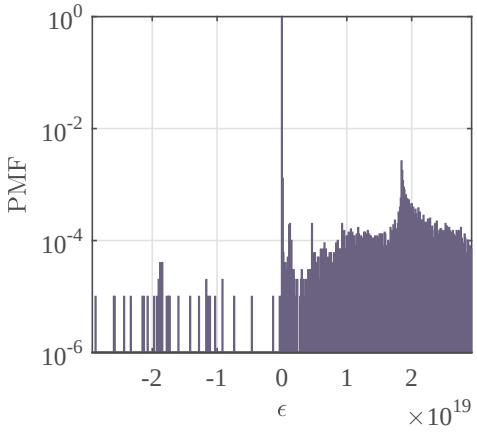
(a) $p = 0.99$.

(b) $p = 0.999$.

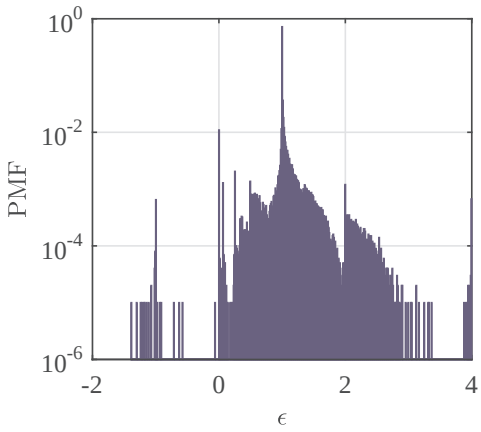


(c) $p = 0.9999$.

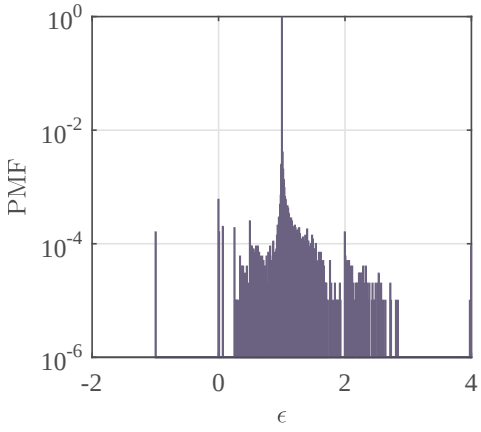
Figure 37. Error histograms for an inexact half-precision floating-point multiplier, with inputs A and B uniformly distributed from -100 to $+100$, for various values of p. Each distribution has several modes, representing the following cases: (1) $\epsilon = 1$ meaning no error; (2) powers of 2, meaning a single bit of the exponent was flipped from 0 to 1; (3) powers of $\frac{1}{2}$, meaning a single bit of the exponent was flipped from 1 to 0; (4) ϵ close to zero, meaning the MSB of the exponent was flipped from 1 to 0; and (5) $\epsilon = -1$, meaning the sign bit was flipped. Results are from Matlab PBL simulation.



(a) $p = 0.99$.



(b) $p = 0.999$.



(c) $p = 0.9999$.

Figure 38. Error histograms for an inexact single-precision floating-point multiplier, with inputs A and B uniformly distributed from -100 to $+100$, for various values of p. Plots are asymmetric because $\epsilon < 0$ only occurs when there is an error in the sign bit of the output. Results are from Matlab PBL simulation.

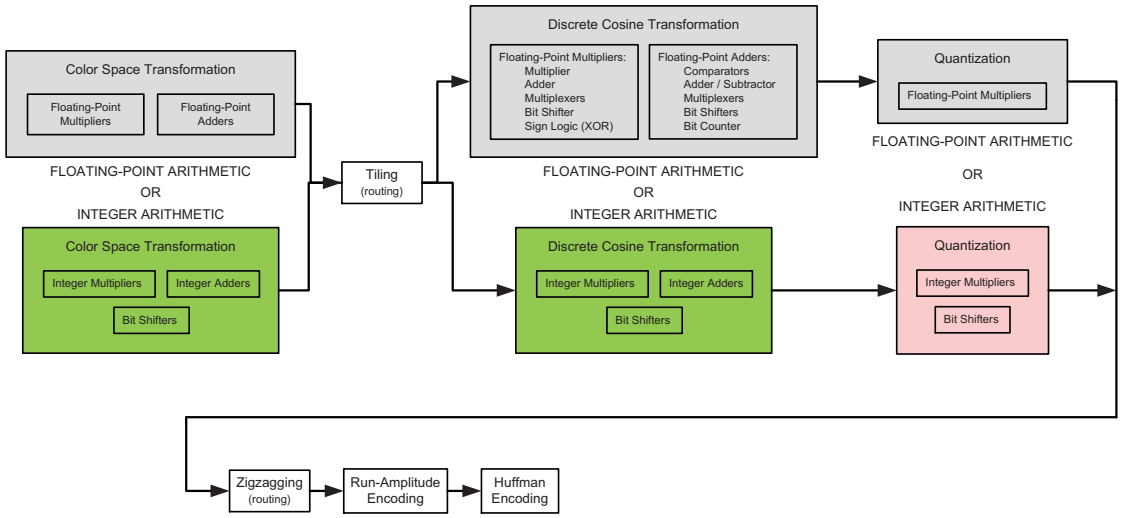


Figure 39. Block diagram of the JPEG image compression algorithm. In this dissertation, the JPEG algorithm is a motivational example for inexact computing. The shaded boxes show areas where inexact methods can be considered (for example, in adder circuits and multiplier circuits). The white boxes show areas where inexact methods cannot be considered ("keep-out zones").

Fig. 40. The images in this figure have not yet undergone compression. The figure shows how the image degrades as p decreases. For example, when $p = 0.999999$ as in Fig. 40b, the RMS error is 0.91 and the SNR is 23.73 dB. If the RMS error is normalized relative to its maximum possible value of 255, then the RMS error of 0.91 normalizes to 0.36%. When $p = 0.99$ as in Fig. 40f, the RMS error degrades to 13.85 (5.4% normalized RMS error) with an SNR of 11.91 dB.

The formula for the CST for the intensity component Y of a single pixel is given in Eq. (69). This equation contains three multiplications and two additions. Using 16-bit multipliers ($N_A = 8, N_B = 8$) with $N_{mult,exact} = 3$, followed by truncation of the lower 8 bits, followed by 8-bit addition with $N_{add,exact} = 3$, we have a total of:

- $2 \times 3 = 6$ exact one-bit adders from the two addition operations,
- $2 \times 5 = 10$ inexact one-bit adders from the two addition operations,
- $3 \times 3 = 9$ exact one-bit adders from the three multiplication operations (see

(a) Original image.

(b) $p = 0.999999$, RMS error $= 0.91$, SNR $= 23.73$ dB.

(c) $p = 0.99999$. RMS error $= 0.99$, SNR $= 23.37$ dB.

(d) $p = 0.9999$. RMS error $= 1.52$, SNR $= 21.51$ dB.

(e) $p = 0.999$. RMS error $= 4.06$, SNR $= 17.24$ dB.

(f) $p = 0.99$. RMS error $= 13.85$, SNR $= 11.91$ dB.

Figure 40. Uncompressed bitmap images computed using an inexact color space transformation with various values of p, RMS error, and signal-to-noise ratio (SNR). Only the intensity component Y is shown.

Table 7), and

- $3 \times 53 = 159$ inexact one-bit adders from the three multiplication operations.

From the above, there are $6 + 9 = 15$ exact one-bit adders and $10 + 159 = 169$ inexact one-bit adders per pixel for the CST. In this example, a contribution of this dissertation is that we show $\frac{169}{169+15} = 91.8\%$ of the CST can be built from inexact components. Using data from Fig. 30, we see that a probability of correctness $p = 0.99$ with a noise PSD of 3×10^{-12} V^2/Hz gives an energy savings of about 62%, and $p = 0.999$ gives an energy savings of about 15%. Employing exact computation on the three most significant bits and using Eq. (107) gives the energy savings and error values shown in Table 11.

Table 11. Energy Savings and Errors for Inexact Color Space Transformation

p	Energy Savings %	Normalized RMS Error %
0.99	57	5.4
0.999	14	1.6

5.4.2 Inexact DCT.

This section illustrates the result of using an inexact DCT, but exact color space transformation and no quantization. An inexact JPEG compression algorithm was simulated as described in Section 4.6 and shown in Fig. 39. Performance results for the inexact DCT are shown in Fig. 41. This figure shows how the image degrades as p decreases. For example, when $p = 0.999999$ as in Fig. 41b, the RMS error is 2.29 (0.90% normalized) and the SNR is 19.73 dB. When $p = 0.99$ as in Fig. 41f, the RMS error degrades to 50.51 (19.8% normalized) with an SNR of 6.29 dB. The artifacts in Fig. 41f resemble those explained in Fig. 15 on page 52; this is the result of errors in the DCT algorithm. The compression ratios are still low (1.63 at best) because

we have not yet done quantization as described in Section 3.6.4. After quantization, typical compression ratios range from 1.7 (using a quality factor $\mathfrak{q} = 100\%$) up to 22 (using a quality factor $\mathfrak{q} = 25\%$) [46, p. 191]. The compression ratio in Fig. 41f is degraded down to 0.72 due to the introduction of artifacts into the DCT as a results of errors in the inexact computation of the DCT. These nonzero artifacts reduce the sparsity of the DCT matrix, resulting in a longer set of run-amplitude encoded data. With proper quantization according to Equation (73), most of those artifacts would be filtered out.

The formula for the discrete cosine transform for an 8×8 block is given in Eq. (72). One 8×8 matrix multiplication contains 512 multiplications and 448 additions, and the second 8×8 matrix multiplication contains another 512 multiplications and 448 additions; the total is 1024 multiplications and 896 additions. Using 16-bit multipliers ($N_A = 8, N_B = 8$) with $N_{mult,exact} = 3$, and 16-bit addition with $N_{add,exact} = 3$, we have a total of:

- $896 \times 3 = 2,688$ exact one-bit adders from the 896 addition operations,
- $896 \times 13 = 11,648$ inexact one-bit adders from the 896 addition operations,
- $1,024 \times 3 = 3,072$ exact one-bit adders from the 1,024 multiplication operations (see Table 7), and
- $1,024 \times 53 = 54,272$ inexact one-bit adders from the 1,024 multiplication operations.

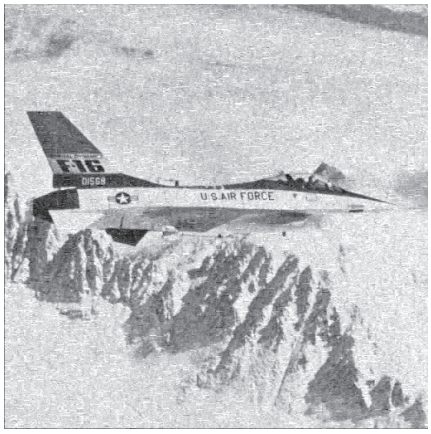
From the above, there are $2,688 + 3,072 = 5,760$ exact one-bit adders and $11,648 + 54,272 = 65,920$ inexact one-bit adders per block for the discrete cosine transformation. In this example, a contribution of this dissertation is that we show $\frac{65,920}{65,920+5,760} = 92.0\%$ of the discrete cosine transform can be built from inexact components. Using data from Fig. 30, we see that a probability of correctness $p = 0.99$ with

(a) Uncompressed image.

(b) $p = 0.999999$, RMS error = 2.29, CR= 1.63, SNR = 19.73 dB.

(c) $p = 0.99999$. RMS error = 2.80, CR = 1.62, SNR = 18.86 dB.

(d) $p = 0.9999$. RMS error = 5.82, CR = 1.47, SNR = 15.68 dB.



(e) $p = 0.999$. RMS error = 17.87, CR = 1.05, SNR = 10.80 dB.

(f) $p = 0.99$. RMS error = 50.51, CR = 0.72, SNR = 6.29 dB.

Figure 41. JPEG images, without quantization, computed using an inexact discrete cosine transformation with various values of p, RMS error, Compression Ratio (CR), and Signal-to-Noise Ratio (SNR). Only the intensity component Y is shown.

a noise PSD of 3×10^{-12} V^2/Hz gives an energy savings of about 62%, and $p = 0.999$ gives an energy savings of about 15%. Employing exact computation on the three most significant bits and using (107) gives the energy savings and error values shown in Table 12. Note that the relative energy savings of the inexact DCT are roughly the same as the relative energy savings for the inexact CST. Future work should further examine optimization of the DCT for energy savings via inexact computing. Future research will also examine the results of performing inexact quantization on the overall performance of the JPEG compression algorithm shown in Fig. 39.

Table 12. Energy Savings and Errors for Inexact Discrete Cosine Transformation

p	Energy Savings %	Normalized RMS Error %
0.99	57	19.8
0.999	14	7.0

5.5 Remarks

Although RMS error is a useful metric of image quality, it is very simplistic. Models of human perception are very complex, and are not easily summarized by simple metrics. If an image is intended for human consumption, it is up to the human observer whether or not the image quality is "good enough".

The JPEG standard has been around since the early 1990s [49]. Although researchers are working on new image compression algorithms, the legacy JPEG algorithm is well-understood and still widely used. The results of this dissertation demonstrate a promising approach to improving other image compression algorithms via inexact computing.

VI. Discussion

The decision flowchart in Fig. 1 provides the overall framework for inexact computing design. In this dissertation, we demonstrated the following steps in the flowchart:

- Profiling of the JPEG algorithm (Fig. 2),
- Deciding which sub-algorithms are amenable to inexact design (Sections 3.6.8 and 4.7 and Fig. 2),
- Choosing the right amount of precision (Section 4.6.1),
- Choosing error metrics (Section 4.7),
- Using noisy inexactness to reduce energy and area (Sections 4.1.1.4, 4.3 and 4.4), and
- Comparing the inexact sub-algorithm design vs. the exact sub-algorithm baseline (Tables 11 and 12).

Tables 11 and 12 illustrate the tradeoffs made within the space of inexact computing design.

With sufficient energy reduction, breakpoints in the tradeoffs begin to emerge. For example, consider the case in which the energy reduction with the use of inexact methods is reduced to 1/4 of the initial energy consumption, as shown in Fig. 32, curve (a), with 73% energy reduction and a 7.5% error standard deviation. In this case, circuit designers can pull ahead of the energy need even when three adders are used (triple module redundancy). In such cases, there are "break points" when energy savings are sufficient to justify a change in circuit architecture, and the contribution

of this dissertation is to provide data to decide the break points for a few specific adder circuits in 14 nm FinFET CMOS technology and 0.6 μm CMOS technology.

This dissertation helps designers find a way to pull ahead, such that designers can find a tradeoff in energy that gives a clear decisive drop in power faster than the growth in area. As an example, consider triple module redundancy for which the breakpoint is approximately 67%, not including the small area overhead in the voting circuitry; thus, designers would need to achieve a gain of approximately four in order to beat an area (overhead) occupied by three modules.

Note that designers can achieve this breakpoint in time or area. In time, one can just take three consecutive samples—which is very attractive for circuits—designers can slow down the circuits, take more samples, and vote; the more one can slow down the circuit, the more accurate the result can be, and then one could achieve perhaps a 20-fold energy reduction. In such a way, designers can play with the time dimension a lot in order to pull ahead.

Another question we are asking is, "What is the best a designer can ever do?" For example, a broken clock is really energy efficient but is not wrong all of the time; the broken clock is correct twice a day. We are trying to understand if it is possible to make a clock that wiggles a bit; in such a case we would expect that it is wrong less of the time. The same principle applies to inexact adders.

Other components, such as barrel shifters, bit counters, multiplexers, multipliers, floating-point adders, and floating-point multipliers can be built using inexact logic circuits. Such components can be used in the JPEG compression algorithm, as shown in Fig. 2. However, in this work we used only integer adders and multipliers, and the only bit shifting we did was simply a matter of truncating some of the least significant bits. The purpose of the compression algorithm is to reduce the size of the data while retaining most of the information contained therein. The first

step in the algorithm is the color space transformation. In this step, the processor transforms the red, blue, and green image components into a luminance component and two chrominance components. This is a linear formula consisting of addition and multiplication, and could be accomplished using integer adders and multipliers. This is a possible application for inexact computing.

The second step of the JPEG compression algorithm, tiling, does not involve inexact components; it consists of wiring only. In this step, the processor arranges each data component into 8×8 blocks of pixels. This is simply routing of data. Under our inexact computing model, routing can be accomplished via hard-wiring without any inexactness.

In the third step, the processor performs the discrete cosine transformation (DCT) on each 8×8 block of data. This consists of two 8×8 matrix multiplications involving non-whole signed numbers. To handle fractional numbers, we chose to use signed integer arithmetic instead of IEEE 754 standard floating point numbers, because initial results indicate integer arithmetic produced lower errors. This signed integer arithmetic was optimized by the use of various p-values, exact computation on the three most significant bits, and precision limited to only the number of bits needed, as described in Section 4.6. Future research will further examine optimization of the DCT for energy and EDP improvement via inexact computing.

The fourth step of the compression algorithm, quantization, consists of dividing (or multiplying) each DCT output by a constant value. The purpose of quantization is to reduce the DCT data into a sparse matrix consisting of mostly zeros. This could also be performed using inexact multipliers.

The final three steps in the compression algorithm are not applications for inexact computing. Step five is zigzagging, which is routing of data, and can be accomplished without inexactness. Step six, which is run-amplitude encoding, and step seven,

which is Huffman encoding, are state machine applications, which are not tolerant of error.

The results shown in this dissertation provide a promising approach to continue improvements on the energy and EDP of the JPEG compression algorithm via inexact computing, at the expense of tradeoffs in image quality and compression ratio. We have demonstrated the inexact color space transformation and DCT; the other component, quantization, is reserved for future work. Future work will also consider the increases in energy consumption caused by degradation of the compression ratio. Also, future research will utilize inexact techniques to optimize JPEG image compression hardware for reduced energy consumption and reduced EDP via inexact computing.

VII. Summary and Conclusions

The past five decades have seen an insatiable demand for computation supported by the success of Moore's Law, and concerns about limits to power reduction that can be realized with traditional digital design and device scaling are being raised. At the same time, a need exists to build a fault tolerant semiconductor chip that can handle failure gracefully. These trends provide motivation for researchers to investigate areas such as probabilistic computing and inexact methods that offer potential improvement in energy (savings), performance (improvement), and area (improvement). Probabilistic computing offers a potential capability to extract functionality that is 'good enough' while operating with lower power dissipation.

This dissertation presents a method to quantify the energy savings resulting from the decision to accept a specified percentage of error in some components of a computing system. With the JPEG algorithm, loss is tolerated in certain components (e.g. color space transformation and discrete cosine transform) that contain adder circuits. The contribution of this dissertation is to provide energy-accuracy tradeoffs for a few inexact adder architectures in 14 nm FinFET CMOS technology.

7.1 Adders

This dissertation investigated the susceptibility to noise of some digital adder circuits that are deliberately engineered to be imprecise. The adders are characterized with probabilistic Boolean logic which provides the capability to characterize random noise in digital circuits. In this study, each binary logic gate was assigned a probability p of correctness between 0.8000 and 0.9999.

The contribution of this dissertation is to provide quantitative data providing energy-accuracy tradeoffs for 8-bit and 16-bit ripple carry adders and 8-bit and 16-

bit Kogge-Stone adders in 14 nm FinFET CMOS technology as functions of four levels of noise power spectral density introduced in the circuits. Error histograms, standard deviation, kurtosis, and probability of zero error are reported. The power supply voltage takes on values in the range of 0.3 V to 0.8 V in 14 nm FinFET CMOS technology. The main results of this dissertation show that the energy reduction can take on the largest value of 92% with an error rate of 0.1 (where the noise power spectral density takes on a value of 3×10^{-12} V^2/Hz). Second, results show that an energy reduction of 95% can be achieved with a standard deviation of a normalized output error of 0.18 (again, where the noise power spectral density takes on a value of 3×10^{-12} V^2/Hz). The results also show that for V_{DD}=0.6 V and p=0.91, energy consumption can be reduced by 50% compared with V_{DD} = 0.8 V, with a normalized error standard deviation of 0.2, and a reduction of 30% in the energy-delay product. When V_{DD} is further reduced to V_{DD} = 0.5 V with p = 0.87, the error standard deviation is 0.25, and energy consumption is reduced by 65%, and energy-delay product is reduced by 40%. Results show that the energy-delay product is minimized when the normalized error standard deviation takes on a value between 0.25 and 0.32. As error increases beyond this point, the increase in delay exceeds the reduction in energy, and so the EDP starts to increase again.

7.2 Multipliers

This dissertation presents a methodology for inexact multipliers, including shift and add, Baugh-Wooley, and Wallace tree multipliers. Probabilistic Boolean logic simulations of these multipliers and the associated error are presented with Matlab. Results show that for a probability of correctness of 0.999, the normalized error standard deviation achieved in an 8-bit shift-and-add multiplier is 2.66%. For the shift-and-add multiplier, a methodology was presented to use exact technology to

compute the most significant bits, and inexact technology to compute the least significant bits, and an example was provided. An expression to calculate the energy per cycle by an exact shift-and-add multiplier was discussed.

The results provided by the Matlab simulations of an 8-bit noisy shift-and-add multiplier and an 8-bit Wallace tree multiplier show that as the probability of correctness p takes on a larger value (increases), the standard deviation in the error decreases in both multipliers, as shown in Fig. 34(a)-(c) and Fig. 35(a)-(c). Similar results are also summarized for 16-bit shift-and-add multipliers and 16-bit Wallace tree multipliers and 32-bit shift-and-add multipliers and 32-bit Wallace tree multipliers, as shown in Fig. 36. Detailed error statistics are summarized in Table 10.

7.3 JPEG

This dissertation presents a methodology for the JPEG compression algorithm. Uncompressed TIFF files from the University of Southern California Signal and Image Processing Institute (SIPI) are evaluated with the methodology presented in this dissertation. This dissertation uses integer arithmetic and Matlab simulations to carry out the inexact JPEG algorithm (See green boxes and pink box in the flowchart in Fig. 39).

The JPEG algorithm is composed of the following steps: color space transformation, tiling, discrete cosine transform, quantization, zigzagging, run-amplitude encoding, and Huffman encoding. As discussed in the dissertation, inexact computing (inexactness) can be tolerated in the first, third, and fourth, steps of the JPEG algorithm, namely color space transformation, discrete cosine transformation, and quantization.

This dissertation presents an inexact approach to the JPEG algorithm through incorporation of inexact adders and inexact multipliers in the color space transfor-

mation step and the discrete cosine transformation step (green boxes in Fig. 39). In this methodology, exact computation is used on the three most significant bits of each addition and multiplication operation in the color space transformation step and in the discrete cosine transformation step. Exact methods are used in the the quantization step for simplicity, even though it is recognized that this step can tolerate inexact methods. Fig. 39 summarizes the JPEG algorithm. Note that floating point arithmetic or integer arithmetic can be used in the methodology.

The results obtained in this dissertation show that a signal-to-noise ratio of 15.68 dB and RMS error of 5.82 can be achieved with a probability of correctness of 99.99%, as shown in Fig. 41(d). The results also show that a signal-to-noise ratio of 6.29 dB and RMS error of 50.51 can be achieved with a probability of correctness of 99%, as shown in Figure 41(f).

We recognize that a fully inexact JPEG algorithm should take advantage of inexact methods at all steps. Therefore future work should consider the incorporation of inexact methods in the quantization step.

7.4 Contributions

Using inexact design methods, we have shown that we could cut energy demand in half with 16-bit Kogge-Stone adders that deviated from the correct value by an average of 3.0 percent in 14 nm CMOS FinFET technology, assuming a noise amplitude of 3×10^{-12} V^2/Hz (see Fig. 32). This was achieved by reducing V_{DD} to 0.6 V instead of its maximum value of 0.8 V. The energy-delay product (EDP) was reduced by 38 percent (see Fig. 33).

Adders that got wrong answers with a larger deviation of about 7.5 percent (using $V_{DD} = 0.5$ V) were up to 3.7 times more energy-efficient, and the EDP was reduced by 45 percent.

Adders that got wrong answers with a larger deviation of about 19 percent (using $V_{DD} = 0.3$ V) were up to 13 times more energy-efficient, and the EDP was reduced by 35 percent.

We used inexact adders and inexact multipliers to perform the color space transform, and found that with a 1 percent probability of error at each logic gate, the letters "F-16", which are 14 pixels tall, and "U.S. AIR FORCE", which are 8 to 10 pixels tall, are readable in the processed image, as shown in Fig. 40f, where the relative RMS error is 5.4 percent.

We used inexact adders and inexact multipliers to perform the discrete cosine transform, and found that with a 1 percent probability of error at each logic gate, the letters "F-16", which are 14 pixels tall, and "U.S. AIR FORCE", which are 8 to 10 pixels tall, are readable in the processed image, as shown in Fig. 41f, where the relative RMS error is 20 percent.

This dissertation demonstrates the implementation of a complex algorithm using inexact design methods. In this demonstration, inexactness is the result of noise, crosstalk, RF interference, cross-chip variations, or other imperfections which affect circuit performance in a probabilistic manner. These results show savings of energy, delay, and area by continuing device scaling with hardware technologies which are less than perfectly reliable. Future research will include fabrication of complex systems using such unreliable technologies, and further development of inexact design methodologies.

Appendix A. Inexact Integer Adders

1.1 Ripple-Carry Adder

```
function [S, Cout, S0] = Adder_RCA_inexact(n, A, B, Cin, p, bit, msbhalfadder)
%Adder_RCA_inexact:  Adds inputs A, B, and Cin, simulating a ripple-carry
%adder, except that each AND, OR, and AO21 (and-or) gate has a random
%error probability equal to 1-p.
%
%Inputs:
%n:  (positive integer) Number of bits processed by the adder.
%A, B:  (n-bit integer arrays) Input arguments for the adder.
%    If B is a scalar, then B is treated as a constant, allowing for a
%    simplified hardware implementation, which results in less inexactness.
%Cin:  (logical array) Carry-in input for the adder.
%p:  (scalar) Probability of correctness of each AND, XOR, and AO21 gate
%    inside the adder.  0 <= p <= 1.
%bit:  (integer vector) Which bit positions can be inexact.  Position 1 is
%    lowest-order bit.  (optional) If bit is omitted, then all positions
%    can be inexact.
%msbhalfadder:  Assumes the most significant bit (MSB) of B is always 0,
%    and uses a half-adder to add the MSB of A and to the carry.
%
%Outputs:
%S0: (2*n-bit integer array) Sum of A, B, and Cin, including carry-out bit.
%S:  (n-bit integer array) Lower n bits of S0, excluding carry-out bit.
%Cout:  (logical array) Carry-out bit.
%
%References:
%N. H. E. Weste and D. M. Harris, CMOS VLSI Design, 4th ed.,
%Boston: Addison-Wesley, 2011, p. 449.
```

```
28  %
29  %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
30  %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
31  %Computer Science, Jun 2008.
32
33  switch class(A)
34      case 'int8'
35          na = 8;     signedA = true;
36      case 'uint8'
37          na = 8;     signedA = false;
38      case 'int16'
39          na = 16;     signedA = true;
40      case 'uint16'
41          na = 16;     signedA = false;
42      case 'int32'
43          na = 32;     signedA = true;
44      case 'uint32'
45          na = 32;     signedA = false;
46      case 'int64'
47          na = 64;     signedA = true;
48      case 'uint64'
49          na = 64;     signedA = false;
50      otherwise
51          error 'Addends must be of the integer classes.'
52  end
53  A = unsigned(A);
54  signA = logical(bitget(A,n));
55
56  switch class(B)
57      case 'int8'
58          nb = 8;     signedB = true;
59      case 'uint8'
60          nb = 8;     signedB = false;
61      case 'int16'
62          nb = 16;     signedB = true;
63      case 'uint16'
64          nb = 16;     signedB = false;
65      case 'int32'
66          nb = 32;     signedB = true;
67      case 'uint32'
68          nb = 32;     signedB = false;
69      case 'int64'
70          nb = 64;     signedB = true;
71      case 'uint64'
```

```
72          nb = 64;     signedB = false;
73      otherwise
74          error 'Addends must be of the integer classes.'
75  end
76  B = unsigned(B);
77  signB = logical(bitget(B,n));
78
79  if n <= 8
80      classname = 'uint8';
81  elseif n <= 16
82      classname = 'uint16';
83  elseif n <= 32
84      classname = 'uint32';
85  else
86      classname = 'uint64';
87  end
88
89  if ~exist('Cin','var')
90      Cin = [];
91  end
92
93  if ~exist('p','var')
94      p = 1;
95  end
96
97  if ~exist('bit','var')
98      bit = 1 : n;
99  end
100
101 if ~exist('msbhalfadder','var')
102     msbhalfadder = false;
103 end
104
105 if isempty(bit)
106     minbit = Inf;
107     maxbit = Inf;
108 else
109     minbit = min(bit(:));
110     maxbit = max(bit(:));
111 end
112
113 constantadder = isscalar(B);
114
115 S = zeros(size(A),classname);
116 Cout = false(size(A));
117
```

```
118 % Initial propagate/generate stage.
119 if constantadder
120     % If we are adding only a constant B, we can use fewer
          components,
121     % and therefore less inexactness.  For each bit:
122     %     A AND 1 = A, and can be omitted altogether from
          hardware.
123     %     A AND 0 = 0.
124     %     A XOR 0 = A, and can be omitted.
125     %     A XOR 1 can be hard-wired as an inverter.
126
127     AandB = bitand(A, B);
                                                    % generate
128     AxorB = bitxor(A, B);
                                                    % propagate
129
130     % Introduce errors only into the bits of B which are
          set high.
131     err = bitand(B, biterrors(size(AxorB), p, classname,
          bit));
132     AxorB = bitxor(AxorB, err);
133     clear err
134 else     % A and B are both variables
135     AandB = bitand_inexact(A, B, p, classname, bit);
                    % generate
136     AxorB = bitxor_inexact(A, B, p, classname, bit);
                    % propagate
137 end
138
139 cols = 1 : n;
140 cols = repmat(cols, [numel(A), 1]);
141 Sbits = false(size(cols));
142 if isempty(Cin)
143     Cbits = [false(numel(A),1), Sbits];
144 elseif isscalar(Cin)
145     Cbits = [repmat(logical(Cin),[numel(A),1]), Sbits];
146 else
147     Cbits = [logical(Cin(:)), Sbits];
148 end
149 Bbits = Sbits;
150 AandBbits = Sbits;
151 AxorBbits = Sbits;
152 AxorBandCbits = Sbits;
153 AandBbits(:) = bitget(repmat(AandB(:),[1,n]),cols);
154 AxorBbits(:) = bitget(repmat(AxorB(:),[1,n]),cols);
155 if isscalar(B)
```

```
156       Bbits(:) = bitget(repmat(B(:),[numel(A),n]),cols);
157   else
158       Bbits(:) = bitget(repmat(B(:),[1,n]),cols);
159   end
160
161   for j = 1 : n
162       k = find(j == bit, 1, 'first');
163       if (j==minbit) && isempty(Cin)                % half
              adder on lowest bit
164           Sbits(:,j) = AxorBbits(:,j);
165           Cbits(:,j+1) = AandBbits(:,j);
166       elseif (j==maxbit) && msbhalfadder            % half
              adder on highest bit
167           Amsb = logical(bitget(A(:),maxbit));
168           Sbits(:,j) = xor(Amsb, Cbits(:,j));
169           Cbits(:,j+1) = Amsb & Cbits(:,j);
170           if k
171               i = (rand(numel(A),1) > p);
172               Sbits(i,j) = ~Sbits(i,j);
173               i = (rand(numel(A),1) > p);
174               Cbits(i,j+1) = ~Cbits(i,j+1);
175           end
176       else                                          % full
              adder
177           Sbits(:,j) = xor(AxorBbits(:,j), Cbits(:,j));
178           if k
179               i = (rand(numel(A),1) > p);
180               Sbits(i,j) = ~Sbits(i,j);
181           end
182           AxorBandCbits(:,j) = AxorBbits(:,j) & Cbits(:,j);
183           if k
184               i = (rand(numel(A),1) > p);
185               AxorBandCbits(i,j) = ~AxorBandCbits(i,j);
186           end
187           Cbits(:,j+1) = AxorBandCbits(:,j) | AandBbits(:,j)
                  ;
188           if k
189               i = (rand(numel(A),1) > p) & (~constantadder |
                      Bbits(:,j));
190               %If B is const, the OR gate is only needed
                      for hi bits of B
191               Cbits(i,j+1) = ~Cbits(i,j+1);
192           end
193       end
194   end
195
```

```
196  Cout(:) = Cbits(:,n+1);
197  twos = cast(pow2(0:(n-1)),'like',S);
198  S(:) = sum(cast(Sbits,'like',S) .* twos(ones(numel(S),1)
       ,:),2,'native');
199
200  if n <= 7
201      S0 = uint8(S);
202  elseif n <= 15
203      S0 = uint16(S);
204  elseif n <= 31
205      S0 = uint32(S);
206  elseif n <= 63
207      S0 = uint64(S);
208  else
209      S0 = double(S) + double(pow2(n) * Cout);
210  end
211
212  if signedA || signedB
213      signout = logical(bitget(S,n));
214  %    overflow = (~signA | ~signout) & (signA | ~signB) & (
       signB | signout);
215
216      0xFFFF = intmax(class(S));
217      0xF000 = bitshift(0xFFFF,n);
218      S(signout) = bitor(S(signout),0xF000);
219      S = signed(S);
220
221      if n <= 63
222          Cout2 = xor(Cout,xor(signA,signB));
223          0xFFFFFFFF = intmax(class(S0));
224          0xF0000000 = bitshift(0xFFFFFFFF,n);
225          S0(Cout2) = bitor(S0(Cout2),0xF0000000);
226          S0 = signed(S0);
227      else
228          S0(signout) = -S0(signout);
229      end
230  elseif n <= 63
231      S0 = bitset(S0,n+1,Cout);
232  end
```

1.2 Kogge-Stone Adder

```
1  function [ S, Cout, S0 ] = kogge_stone_inexact_PBL( n, A,
       B, Cin, p )
2  %kogge_stone_inexact_PBL:  Adds inputs A, B, and Cin,
       simulating a Kogge-
3  %Stone adder, except that each AND, XOR, and AO21 (and-or)
       gate has a
4  %random error probability equal to 1-p.
5  %
6  %Inputs:
7  %n:  (positive integer) Number of bits processed by the
       adder.
8  %A, B:  (n-bit integer arrays) Input arguments for the
       adder.
9  %    A, B, and Cin must all have the same dimensions.
10 %Cin:  (logical array) Carry-in input for the adder.
11 %p:  (scalar) Probability of correctness of each AND, XOR,
       and AO21 gate
12 %    inside the adder.  0 <= p <= 1.
13 %
14 %Outputs:
15 %S0: (2*n-bit integer array) Sum of A, B, and Cin,
       including carry-out bit.
16 %S:  (n-bit integer array) Lower n bits of S0, excluding
       carry-out bit.
17 %Cout:  (logical array) Carry-out bit.
18 %
19 %References:
20 %N. H. E. Weste and D. M. Harris, CMOS VLSI Design, 4th ed
       .,
21 %Boston: Addison-Wesley, 2011, p. 449.
22 %
23 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
24 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
25 %Computer Science, Jun 2008.
26
27 if n <= 7
28     classname0 = 'uint8';
29     classname = 'uint8';
30 elseif n <= 8
31     classname0 = 'uint8';
32     classname = 'uint16';
33 elseif n <= 15
```

```
34      classname0 = 'uint16';
35      classname = 'uint16';
36  elseif n <= 16
37      classname0 = 'uint16';
38      classname = 'uint32';
39  elseif n <= 31
40      classname0 = 'uint32';
41      classname = 'uint32';
42  elseif n <= 32
43      classname0 = 'uint32';
44      classname = 'uint64';
45  else
46      classname0 = 'uint64';
47      classname = 'uint64';
48  end
49
50  if nargin < 5
51      p = 1;
52  end
53
54  if (nargin < 4) || isempty(Cin)
55      c = 0;
56      Cin = zeros(classname);
57  else
58      c = 1;
59      Cin0 = logical(Cin);
60      Cin = zeros(size(Cin),classname);
61      Cin(:) = Cin0;
62  end
63
64  logn = log2(n) + c + 1;
65  S0 = zeros([max([numel(A), numel(B)]), 1], classname);
66  P = zeros([max([numel(A), numel(B)]), logn], classname);
67  G = P;
68
69  P(:,1) = bitxor_inexact(A(:), B(:), p, classname0, 1:n);
70  G(:,1) = bitand_inexact(A(:), B(:), p, classname0, 1:n);
71  if c
72      P(:,1) = bitshift(P(:,1), 1);
73      G(:,1) = bitor(bitshift(G(:,1), 1), Cin(:));
74  end
75
76  i2 = zeros(classname);
77  for i = 2 : logn
78      i1 = pow2(i-2);
79      i2(:) = pow2(i1)-1;
```

```
80      i2c = bitcmp0(i2, n+c);
81      P(:,i) = bitand_inexact(P(:,i-1), bitshift(P(:,i-1),i1
            ), p, classname, (2*i1+1):(n+c+1));
82      P(:,i) = bitor(bitand(P(:,i),i2c), bitand(P(:,i-1),i2)
            );
83      G(:,i) = AO21_inexact(G(:,i-1), P(:,i-1), bitshift(G
            (:,i-1),i1), p, classname, (i1+1):(n+c+1));
84      G(:,i) = bitor(bitand(G(:,i),i2c), bitand(G(:,i-1),i2)
            );
85  end
86
87  S0(:) = bitxor_inexact(P(:,1), bitshift(G(:,logn),1), p,
        classname, 2:(n+c+1));
88  if c
89      S0(:) = bitshift(S0(:), -1);
90  end
91
92  if isscalar(A)
93      S0 = reshape(S0, size(B));
94  else
95      S0 = reshape(S0, size(A));
96  end
97
98  intmax1 = zeros(classname);
99  intmax1(:) = pow2(n) - 1;
100 S = zeros(size(S0), classname0);
101 S(:) = bitand(S0, intmax1);
102
103 Cout = false(size(S0));
104 Cout(:) = bitget(S0, n+1);
```

1.3 Ling Radix-4 Carry-Lookahead Adder

```
1  function [ S, Cout, S0 ] = ling_adder_inexact_PBL( n, A, B
       , Cin, p )
2  %ling_adder:  Adds inputs A, B, and Cin, simulating a
       valency-4
3  %carry lookahead adder using the Ling technique.
4  %
5  %Inputs:
6  %n:  (positive integer) Number of bits processed by the
       adder.
7  %A, B:  (n-bit integer arrays) Input arguments for the
       adder.
8  %    A, B, and Cin must all have the same dimensions.
9  %Cin:  (logical array) Carry-in input for the adder.
10 %p:  (scalar) Probability of correctness of each AND, OR,
       XOR, and AOAO2111
11 %    gate inside the adder.  0 <= p <= 1.
12 %
13 %Outputs:
14 %S0: (2*n-bit integer array) Sum of A, B, and Cin,
       including carry-out bit.
15 %S:  (n-bit integer array) Lower n bits of S0, excluding
       carry-out bit.
16 %Cout:  (logical array) Carry-out bit.
17 %
18 %Reference:
19 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
20 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
21 %Computer Science, Jun 2008.
22
23 switch class(A)
24     case {'int8', 'uint8'}
25         na = 8;
26     case {'int16', 'uint16'}
27         na = 16;
28     case {'int32', 'uint32'}
29         na = 32;
30     case {'int64', 'uint64'}
31         error '64-bit␣not␣supported.'
32     otherwise
33         error 'Addends␣must␣be␣of␣the␣integer␣classes.'
34 end
35
```

```
36  switch class(B)
37      case {'int8', 'uint8'}
38          nb = 8;
39      case {'int16', 'uint16'}
40          nb = 16;
41      case {'int32', 'uint32'}
42          nb = 32;
43      case {'int64', 'uint64'}
44          error '64-bit not supported.'
45      otherwise
46          error 'Addends must be of the integer classes.'
47  end
48
49  if n <= 7
50      classname0 = 'uint8';
51      classname = 'uint8';
52  elseif n <= 8
53      classname0 = 'uint8';
54      classname = 'uint16';
55  elseif n <= 15
56      classname0 = 'uint16';
57      classname = 'uint16';
58  elseif n <= 16
59      classname0 = 'uint16';
60      classname = 'uint32';
61  elseif n <= 31
62      classname0 = 'uint32';
63      classname = 'uint32';
64  elseif n <= 32
65      classname0 = 'uint32';
66      classname = 'uint64';
67  else
68      classname0 = 'uint64';
69      classname = 'uint64';
70  end
71
72  if nargin < 5
73      p = 1;
74  end
75
76  if (nargin < 4) || isempty(Cin)
77      c = 0;
78      Cin = zeros(classname);
79  else
80      c = 1;
81      Cin0 = logical(Cin);
```

```
82      Cin = zeros(size(Cin),classname);
83      Cin(:) = Cin0;
84  end
85
86  L = ceil(n / 4);
87  S0 = zeros([max([numel(A), numel(B)]), 1], classname);
88  H = zeros([max([numel(A), numel(B)]), 2], classname);
89  P = zeros([max([numel(A), numel(B)]), 1], classname);
90  I = P;
91
92  P(:,1) = bitxor_inexact(A(:), B(:), p, classname0, 1:n);
93  I(:,1) = bitor_inexact(A(:), B(:), p, classname0, 1:n);
94  H(:,1) = bitand_inexact(A(:), B(:), p, classname0, 1:n);
95  P(:,1) = bitshift(P(:,1), 1);
96  I(:,1) = bitshift(I(:,1), 1);
97  H(:,1) = bitshift(H(:,1), 1);
98  if c
99      H(:,1) = bitor(H(:,1), Cin(:));
100     I(:,1) = bitor(I(:,1), Cin(:));
101 end
102 I(:,1) = bitshift(I(:,1), 1);
103 H(:,2) = H(:,1);
104 I(:,2) = I(:,1);
105
106 for i = 1 : L
107     H(:,1) = bitset(H(:,1), 4*i+1, bitor_inexact( ...
108         bitget(H(:,1), 4*i+1), AOAO2111_inexact( ...
109         bitget(H(:,1), 4*i), bitget(I(:,1), 4*i+1), ...
110         bitget(H(:,1), 4*i-1), bitget(I(:,1), 4*i), ...
111         bitget(H(:,1), 4*i-2), p, classname, 1), p,
                classname, 1));
112     I(:,1) = bitset(I(:,1), 4*i+1, bitand4_inexact( ...
113         bitget(I(:,1), 4*i+1), ...
114         bitget(I(:,1), 4*i), ...
115         bitget(I(:,1), 4*i-1), ...
116         bitget(I(:,1), 4*i-2), p, classname, 1));
117     H(:,1) = bitset(H(:,1), 4*i+1, AO21_inexact( ...
118         bitget(H(:,1), 4*i+1), bitget(I(:,1), 4*i+1), ...
119         bitget(H(:,1), 4*i-3), p, classname, 1));
120     H(:,1) = bitset(H(:,1), 4*i-2, AO21_inexact( ...
121         bitget(H(:,1), 4*i-2), bitget(I(:,1), 4*i-2), ...
122         bitget(H(:,1), 4*i-3), p, classname, 1));
123     H(:,1) = bitset(H(:,1), 4*i-1, AO21_inexact( ...
124         bitget(H(:,1), 4*i-1), bitget(I(:,1), 4*i-1), ...
125         bitget(H(:,1), 4*i-2), p, classname, 1));
126     H(:,1) = bitset(H(:,1), 4*i, AO21_inexact( ...
```

```
127             bitget(H(:,1), 4*i), bitget(I(:,1), 4*i), ...
128             bitget(H(:,1), 4*i-1), p, classname, 1));
129     end
130
131     S0(:) = mux2_inexact(bitshift(H(:,1),1), P(:,1), bitxor(P
            (:,1), I(:,2)), ...
132         p, classname, 1:(n+2));
133     S0(:) = bitshift(S0(:), -1);
134
135     if isscalar(A)
136         S0 = reshape(S0, size(B));
137     else
138         S0 = reshape(S0, size(A));
139     end
140
141     intmax1 = zeros(classname);
142     intmax1(:) = pow2(n) - 1;
143     S = zeros(size(S0), classname0);
144     S(:) = bitand(S0, intmax1);
145
146     Cout = false(size(S0));
147     Cout(:) = bitget(S0, n+1);
```

1.4 Brent-Kung Adder

```
1  function [ S, Cout, S0 ] = brent_kung_inexact_PBL( n, A, B
       , Cin, p )
2  %brent_kung_inexact_PBL:  Adds inputs A, B, and Cin,
       simulating a Brent-Kung
3  %adder, except that each AND, XOR, and AO21 (and-or) gate
       has a random error
4  %probability equal to 1-p.
5  %
6  %Inputs:
7  %n:  (positive integer) Number of bits processed by the
       adder.
8  %A, B:  (n-bit integer arrays) Input arguments for the
       adder.
9  %    A, B, and Cin must all have the same dimensions.
10 %Cin:  (logical array) Carry-in input for the adder.
11 %p:  (scalar) Probability of correctness of each AND, XOR,
        and AO21 gate
12 %    inside the adder.  0 <= p <= 1.
13 %
14 %Outputs:
15 %S0: (2*n-bit integer array) Sum of A, B, and Cin,
       including carry-out bit.
16 %S:  (n-bit integer array) Lower n bits of S0, excluding
       carry-out bit.
17 %Cout:  (logical array) Carry-out bit.
18 %
19 %References:
20 %N. H. E. Weste and D. M. Harris, CMOS VLSI Design, 4th ed
       .,
21 %Boston: Addison-Wesley, 2011, p. 449.
22 %
23 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
24 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
25 %Computer Science, Jun 2008.
26
27 switch class(A)
28     case {'int8', 'uint8'}
29         na = 8;
30     case {'int16', 'uint16'}
31         na = 16;
32     case {'int32', 'uint32'}
33         na = 32;
```

```
34      case {'int64', 'uint64'}
35          error '64-bit␣not␣supported.'
36      otherwise
37          error 'Addends␣must␣be␣of␣the␣integer␣classes.'
38  end
39
40  switch class(B)
41      case {'int8', 'uint8'}
42          nb = 8;
43      case {'int16', 'uint16'}
44          nb = 16;
45      case {'int32', 'uint32'}
46          nb = 32;
47      case {'int64', 'uint64'}
48          error '64-bit␣not␣supported.'
49      otherwise
50          error 'Addends␣must␣be␣of␣the␣integer␣classes.'
51  end
52
53  if n <= 7
54      classname0 = 'uint8';
55      classname = 'uint8';
56  elseif n <= 8
57      classname0 = 'uint8';
58      classname = 'uint16';
59  elseif n <= 15
60      classname0 = 'uint16';
61      classname = 'uint16';
62  elseif n <= 16
63      classname0 = 'uint16';
64      classname = 'uint32';
65  elseif n <= 31
66      classname0 = 'uint32';
67      classname = 'uint32';
68  elseif n <= 32
69      classname0 = 'uint32';
70      classname = 'uint64';
71  else
72      classname0 = 'uint64';
73      classname = 'uint64';
74  end
75
76  if nargin < 5
77      p = 1;
78  end
79
```

```
80  if (nargin < 4) || isempty(Cin)
81      c = 0;
82      Cin = zeros(classname);
83  else
84      c = 1;
85      Cin0 = logical(Cin);
86      Cin = zeros(size(Cin),classname);
87      Cin(:) = Cin0;
88      clear Cin0
89  end
90
91  logn = 2 * log2(n);
92  S0 = zeros([max([numel(A), numel(B)]), 1], classname);
93  P = zeros([max([numel(A), numel(B)]), logn], classname);
94  G = P;
95
96  P(:,1) = bitxor_inexact(A(:), B(:), p, classname0, 1:n);
97  G(:,1) = bitand_inexact(A(:), B(:), p, classname0, 1:n);
98  if c
99      P(:,1) = bitshift(P(:,1), 1);
100     G(:,1) = bitor(bitshift(G(:,1), 1), Cin(:));
101 end
102
103 i2c = zeros(classname);
104 i3c = i2c;
105 for i = 2 : (0.5 * logn + 1)
106     i1 = pow2(i-2);
107     i2c(:) = sum(pow2((pow2(i)-1):pow2(i-1):(n+c-1)));
108     i3c(:) = sum(pow2((pow2(i-1)-1):pow2(i-1):(n+c-1)));
109     i2 = bitcmp(i2c, n+c);
110     i3 = bitcmp(i3c, n+c);
111     P(:,i) = bitand_inexact(P(:,i-1), bitshift(P(:,i-1),i1
            ), p, classname);
112     P(:,i) = bitor(bitand(P(:,i),i2c), bitand(P(:,i-1),i2)
            );
113     G(:,i) = AO21_inexact(G(:,i-1), P(:,i-1), bitshift(G
            (:,i-1),i1), p, classname);
114     G(:,i) = bitor(bitand(G(:,i),i3c), bitand(G(:,i-1),i3)
            );
115 end
116
117 for i = (0.5 * logn + 2) : logn
118     i1 = pow2(logn-i);
119     i3c(:) = sum(pow2((3*pow2(logn-i)-1):(pow2(logn-i+1))
            :(n+c-1)));
120     i3 = bitcmp(i3c, n+c);
```

```
121       P(:,i) = P(:,i-1);
122       G(:,i) = AO21_inexact(G(:,i-1), P(:,i-1), bitshift(G
              (:,i-1),i1), p, classname);
123       G(:,i) = bitor(bitand(G(:,i),i3c), bitand(G(:,i-1),i3)
              );
124   end
125
126   S0(:) = bitxor_inexact(P(:,1), bitshift(G(:,logn),1), p,
          classname, 2:(n+c+1));
127   if c
128       S0(:) = bitshift(S0(:), -1);
129   end
130
131   if isscalar(A)
132       S0 = reshape(S0, size(B));
133   else
134       S0 = reshape(S0, size(A));
135   end
136
137   intmax1 = zeros(classname);
138   intmax1(:) = pow2(n) - 1;
139   S = zeros(size(S0), classname0);
140   S(:) = bitand(S0, intmax1);
141
142   Cout = false(size(S0));
143   Cout(:) = bitget(S0, n+1);
```

1.5 Sklansky Adder

```
1  function [ S, Cout, S0 ] = sklansky_inexact_PBL( n, A, B,
       Cin, p )
2  %sklansky_inexact_PBL:  Adds inputs A, B, and Cin,
       simulating a Sklansky
3  %adder, except that each AND, XOR, and AO21 (and-or) gate
       has a random
4  %error probability equal to 1-p.
5  %
6  %Inputs:
7  %n:  (positive integer) Number of bits processed by the
       adder.
8  %A, B:  (n-bit integer arrays) Input arguments for the
       adder.
9  %    A, B, and Cin must all have the same dimensions.
10 %Cin:  (logical array) Carry-in input for the adder.
11 %p:  (scalar) Probability of correctness of each AND, XOR,
        and AO21 gate
12 %    inside the adder.  0 <= p <= 1.
13 %
14 %Outputs:
15 %S0: (2*n-bit integer array) Sum of A, B, and Cin,
       including carry-out bit.
16 %S:  (n-bit integer array) Lower n bits of S0, excluding
       carry-out bit.
17 %Cout:  (logical array) Carry-out bit.
18 %
19 %References:
20 %N. H. E. Weste and D. M. Harris, CMOS VLSI Design, 4th ed
       .,
21 %Boston: Addison-Wesley, 2011, p. 449.
22 %
23 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
24 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
25 %Computer Science, Jun 2008.
26
27 if n <= 7
28     classname0 = 'uint8';
29     classname = 'uint8';
30 elseif n <= 8
31     classname0 = 'uint8';
32     classname = 'uint16';
33 elseif n <= 15
```

```
34      classname0 = 'uint16';
35      classname = 'uint16';
36  elseif n <= 16
37      classname0 = 'uint16';
38      classname = 'uint32';
39  elseif n <= 31
40      classname0 = 'uint32';
41      classname = 'uint32';
42  elseif n <= 32
43      classname0 = 'uint32';
44      classname = 'uint64';
45  else
46      classname0 = 'uint64';
47      classname = 'uint64';
48  end
49
50  if (~exist('p','var')) || isempty(p)
51      p = 1;
52  end
53
54  if (~exist('Cin','var')) || isempty(Cin)
55      c = 0;
56      Cin = zeros(classname);
57  else
58      c = 1;
59      Cin0 = logical(Cin);
60      Cin = zeros(size(Cin),classname);
61      Cin(:) = Cin0;
62  end
63
64  logn = log2(n) + c + 1;
65  S0 = zeros([max([numel(A), numel(B)]), 1], classname);
66  P = zeros([max([numel(A), numel(B)]), logn], classname);
67  G = P;
68  bits0 = 1 : pow2(logn - 1);
69
70  P(:,1) = bitxor_inexact(A(:), B(:), p, classname0, 1:n);
71  G(:,1) = bitand_inexact(A(:), B(:), p, classname0, 1:n);
72  if c
73      P(:,1) = bitshift(P(:,1), 1);
74      G(:,1) = bitor(bitshift(G(:,1), 1), Cin(:));
75  end
76
77  for i = 2 : logn
78      i1 = pow2(i-2);
79      bits0 = reshape(bits0, [2*i1, numel(bits0)/(2*i1)]);
```

```
80      bits1 = bitset(zeros(classname), bits0(1:i1,:),
           classname);
81      bits1 = sum(bits1(:),'native');
82      bits1c = bitcmp0(bits1, n+c);
83      bits1c_ = bits0((i1+1):end,:);
84      bits2 = bitset(zeros(classname), i1 : (2*i1) : (2*pow2
           (logn-i)*i1-i1), classname);
85      bits2 = sum(bits2(:),'native');
86      P1 = bitand(P(:,i-1), bits1);
87      G1 = bitand(G(:,i-1), bits1);
88      P1c = bitand(P(:,i-1), bits1c);
89      G1c = bitand(G(:,i-1), bits1c);
90      P2_ = bitand(P(:,i-1), bits2);
91      G2_ = bitand(G(:,i-1), bits2);
92      P2 = P2_;
93      G2 = G2_;
94      for j = 1 : (i1-1)
95          P2 = P2_ + bitshift(P2,1);
96          G2 = G2_ + bitshift(G2,1);
97      end
98      P2 = bitshift(P2,1);
99      G2 = bitshift(G2,1);
100     P(:,i) = bitor(P1, bitand_inexact(P1c, P2, p,
           classname, bits1c_));
101     G(:,i) = bitor(G1, AO21_inexact(G1c, P1c, G2, p,
           classname, bits1c_));
102 end
103
104 S0(:) = bitxor_inexact(P(:,1), bitshift(G(:,logn),1), p,
       classname, 2:(n+c+1));
105 if c
106     S0(:) = bitshift(S0(:), -1);
107 end
108
109 if isscalar(A)
110     S0 = reshape(S0, size(B));
111 else
112     S0 = reshape(S0, size(A));
113 end
114
115 intmax1 = zeros(classname);
116 intmax1(:) = intmax(classname0);
117 S = zeros(size(S0), classname0);
118 S(:) = bitand(S0, intmax1);
119
120 Cout = false(size(S0));
```

```
121   Cout(:) = bitget(S0, n+1);
```

1.6 Adder Front-End

The following is a front-end for all of the inexact integer adders.

```
1  function [ S, Cout, S0 ] = adder_inexact_PBL( arch, n, A,
       B, Cin, p )
2  %adder_inexact_PBL:  Adds inputs A, B, and Cin, simulating
       an
3  %adder, except that each AND, XOR, and AO21 (and-or) gate
       has a random
4  %error probability equal to 1-p.
5  %
6  %Inputs:
7  %arch:  (string) Inexact adder architecture:  'RC' (ripple
       -carry),
8  %    'Ling' (valency-4 carry lookahead), 'Sklansky', '
       Brent-Kung', or
9  %    'Kogge-Stone'.
10 %n:  (positive integer) Number of bits processed by the
       adder.
11 %A, B:  (n-bit integer arrays) Input arguments for the
       adder.
12 %    A, B, and Cin must all have the same dimensions.
13 %Cin:  (logical array) Carry-in input for the adder.
14 %p:  (scalar) Probability of correctness of each AND, OR,
       XOR, AO21, and/or
15 %    AOAO2111 gate inside the adder.  0 <= p <= 1.
16 %
17 %Outputs:
18 %S0: (2*n-bit integer array) Sum of A, B, and Cin,
       including carry-out bit.
19 %S:  (n-bit integer array) Lower n bits of S0, excluding
       carry-out bit.
20 %Cout:  (logical array) Carry-out bit.
21 %
22 %References:
23 %N. H. E. Weste and D. M. Harris, CMOS VLSI Design, 4th ed
       .,
24 %Boston: Addison-Wesley, 2011, p. 449.
25 %
26 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
27 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
28 %Computer Science, Jun 2008.
29
```

```
30  if (~exist('arch','var')) || isempty(arch)
31      arch = 'RC';
32  end
33
34  if ~exist('Cin','var')
35      Cin = [];
36  end
37
38  if (~exist('p','var')) || isempty(p)
39      p = 1;
40  end
41
42  curdir = pwd;
43  dirname = userpath;
44  if (dirname(end) == ';') || (dirname(end) == '\')
45      dirname = dirname(1:end-1);
46  end
47
48  switch upper(arch)
49      case {'LING','LING␣CLA','LING-CLA','CLA','CARRY␣
           LOOKAHEAD','CARRY-LOOKAHEAD','CARRY␣LOOK␣AHEAD'}
50          dirname = [dirname, '\Ling-CLA'];
51          fname = 'ling_adder_inexact_PBL';
52      case 'SKLANSKY'
53          dirname = [dirname, '\Sklansky'];
54          fname = 'sklansky_inexact_PBL';
55      case {'BRENT-KUNG','BRENT␣KUNG'}
56          dirname = [dirname, '\Brent-Kung'];
57          fname = 'brent_kung_inexact_PBL';
58      case {'KOGGE-STONE','KOGGE␣STONE'}
59          dirname = [dirname, '\Kogge-Stone'];
60          fname = 'kogge_stone_inexact_PBL';
61      otherwise     % use inexact ripple-carry architecture
62          dirname = [dirname, '\Ripple-Carry'];
63          fname = 'Adder_RCA_inexact';
64  end
65
66  cd(dirname)
67
68  try
69      [S, Cout, S0] = feval(fname, n, A, B, Cin, p);
70  catch exception
71      cd(curdir)
72      rethrow(exception)
73  end
74
```

```
75  cd(curdir)
```

1.7 Adder-Subtractor

```
function [S, Cout, S0] = adder_subtractor_inexact_PBL(arch, n, A, B, D, p)
%adder_subtractor_inexact_PBL adds or subtracts inputs A, B,
%depending on the value of D, simulating an inexact digital
%adder-subtractor.  Each AND, XOR, NOT, and AO21 (and-or)
%gate within the circuit has a random error probability
%equal to 1-p.
%
%Inputs:
%arch:  (string) Inexact adder architecture:  'RC' (ripple-
%    carry), 'Ling' (valency-4 carry lookahead), 'Sklansky',
%    'Brent-Kung', or 'Kogge-Stone'.
%n: (positive integer) Number of bits processed by the adder
%A, B: (n-bit integer arrays) Input arguments for the adder.
%    A, B, and D must all have the same dimensions.
%D:  (logical array) Determines whether the device performs
%    addition or subtraction.  If D is false, then add.  If
%    D is true, then subtract.
%p:  (scalar) Probability of correctness of each AND, OR,
%    NOT, XOR, AO21, and/or AOAO2111 gate inside the adder.
%    0 <= p <= 1.
%
%Outputs:
%S0:  (2*n-bit integer array) Sum A+B or difference A-B,
%    including the carry-out bit.
%S:  (n-bit integer array) Lower n bits of S0, excluding the
%    carry-out bit.
%Cout:  (logical array) Carry-out bit.  For addition, Cout
%    is true in case of a carry.  For subtraction, Cout is
%    false in case of a borrow.
%
%References:
%N. H. E. Weste and D. M. Harris, CMOS VLSI Design, 4th ed.,
```

```
33  %Boston: Addison-Wesley, 2011, p. 449.
34  %
35  %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
36  %Boolean logic and its meaning," Tech. Rep. TR-08-05, Rice
37  %University, Department of Computer Science, Jun 2008.
38
39  if (~exist('arch','var')) || isempty(arch)
40      arch = 'RC';
41  end
42
43  if ~exist('D','var') || isempty(D)
44      D = false;
45  end
46
47  if (~exist('p','var')) || isempty(p)
48      p = 1;
49  end
50
51  OxFFFF = bitcmp0(zeros('like',B),n);
52  B_ = bitcmp_inexact(B,n,p);
53  D1 = cast(D,'like',B) * OxFFFF;
54  B1 = mux2_inexact(D1,B,B_,p);
55
56  [S, Cout, S0] = adder_inexact_PBL(arch, n, A, B1, D, p);
57  Cout_ = xor_inexact(Cout,D,p);
58  S0 = bitset(cast(S,'like',S0),n+1,Cout_);
```

Appendix B. Inexact Floating-Point Adder

```
function [ Ss, Es, Ms ] = Adder_floating_inexact( Sa, Ea,
   Ma, Sb, Eb, Mb, fmt, p )

switch upper(fmt)
    case 'BINARY16'
        ne = 5;
        nm = 10;
    case 'BINARY32'
        ne = 8;
        nm = 23;
    case 'BINARY64'
        ne = 11;
        nm = 52;
    case 'BINARY128'
        ne = 15;
        nm = 112;
    otherwise
        error 'fmt must be binary16, binary32, binary64, or binary128.'
end

if ~exist('p','var')
    p = 1;
end

[Sa1,Ea1,Ma1,~,Eb1,Mb1,D] = big_comparator(Sa,Ea,Ma,Sb,Eb,
   Mb,ne,nm,p);

Ediff = adder_subtractor_inexact_PBL('RC',ne,Ea1,Eb1,true,
   p);   %abs(Ea-Eb)

[Ma2,nm4] = prepend_mantissa(Ea1,Ma1,ne,nm,p);
Mb2 = prepend_mantissa(Eb1,Mb1,ne,nm,p);

Mb3 = rightshift_mantissa(Mb2,Ediff,nm4,ne,p);

OxFF = cast(pow2a(ne,'uint64') - 1, 'like', Ea);
Ox7FFFFF = cast(pow2a(nm4,'uint64') - 1, 'like', Ma2);

Ss = Sa1;
[~,Mc,Ms1] = adder_subtractor_inexact_PBL('RC',nm4,Ma2,Mb3
   ,D,p);      % add
Mc1 = and_inexact(Mc,~D,p);
```

```
39  Ea2 = adder_inexact_PBL('RC',ne,Ea1,zeros('like',Ea1),Mc1,
       p);   %carry to exponent
40  [~,~,Ms2] = bitshifter_inexact_PBL(Ms1,-int8(Mc1),nm4+1,1,
       p);   %if carry-out, then shift right
41
42  Eborrow_ = cast(bitcounter(Ms2,nm4,p), 'like', Ea);
43  Eborrow = bitcmp(Eborrow_);
44  gd_ = comparator_inexact_PBL(ne,Ea2,Eborrow,p);   %
       graceful degradation (GD) flag
45  gd__ = cast(gd_,'like',Ea) * 0xFF;
46  gd___ = cast(gd_,'like',Ma2) * 0x7FFFFF;
47  Ea3 = adder_inexact_PBL('RC',ne,Ea2,Eborrow_,true,p);  %
       borrow from exponent
48  Ms3 = leftshift_mantissa(Ms2,nm4,p);          % shift left
       after subtraction
49
50  Ms3g = bitshifter_inexact_PBL(Ms2,Ea2,nm4,ne,p);   %
       mantissa in case of GD
51  Ms4 = mux2_inexact(gd___,Ms3g,Ms3,p,class(Ms3),1:nm4);
52
53  R = roundoff(Ms4,p);
54  Ms5 = bitshift(bitset(Ms4,nm4,0),-3);       % drop extra
       bits from mantissa
55  [Ms,Mc2] = adder_inexact_PBL('RC',nm,Ms5,zeros('like',Ms5)
       ,R,p);    %round off
56
57  Ea4 = adder_inexact_PBL('RC',ne,Ea3,zeros('like',Ea3),Mc2,
       p);
58  Es = mux2_inexact(gd__,zeros('like',Ea4),Ea4,p,class(Ea4)
       ,1:ne);   % with GD, exponent=0
59
60  end
61
62  function [ Sa1, Ea1, Ma1, Sb1, Eb1, Mb1, D ] =
       big_comparator( Sa, Ea, Ma, Sb, Eb, Mb, ne, nm, p )
63  %Compares the magnitude of input A with the magnitude of
       input B.  If A>B,
64  %then output A1=A and output B1=B.  If A<=B, then A1=B and
        B1=A.
65
66  0xFF = cast(pow2a(ne,'uint64') - 1, 'like', Ea);
67  0x7FFFFF = cast(pow2a(nm,'uint64') - 1, 'like', Ma);
68  n = ne + nm;
69  classname = classn(n);
70  Ea_ = cast(Ea,classname);
71  Ma_ = cast(Ma,classname);
```

```
72  Eb_ = cast(Eb,classname);
73  Mb_ = cast(Mb,classname);
74  AA = bitshift(Ea_,nm) + Ma_;                    % merge
       mantissa with exponent
75  BB = bitshift(Eb_,nm) + Mb_;
76  [AgtB,~,AeqB] = comparator_inexact_PBL(n,AA,BB,p);
                     % compare
77  AgtB_ = cast(AgtB,'like',Ea) * 0xFF;
78  AgtB__ = cast(AgtB,'like',Ma) * 0x7FFFFF;
79
80  D = xor_inexact(Sa,Sb,p);
81  Z = and_inexact(D,AeqB,p);
82  Z_ = cast(Z,'like',Ea) * 0xFF;
83  Z__ = cast(Z,'like',Ma) * 0x7FFFFF;
84  Saz = mux2_inexact(Z,Sa,false,p,class(Sa),1);
85  Eaz = mux2_inexact(Z_,Ea,zeros('like',Ea),p,class(Ea),1:ne
       );   % if B == -A, then output 0
86  Maz = mux2_inexact(Z__,Ma,zeros('like',Ma),p,class(Ma),1:
       nm);
87  Sbz = mux2_inexact(Z,Sb,false,p,class(Sb),1);
88  Ebz = mux2_inexact(Z_,Eb,zeros('like',Eb),p,class(Eb),1:ne
       );   % if B == -A, then output 0
89  Mbz = mux2_inexact(Z__,Mb,zeros('like',Mb),p,class(Mb),1:
       nm);
90
91  [Sa1,Sb1] = mux2_inexact(AgtB,Sbz,Saz,p,class(Sa),1);
             % assign the greater value to A1
92  [Ea1,Eb1] = mux2_inexact(AgtB_,Ebz,Eaz,p,class(Ea),1:ne);
         %   and the lesser value to B1
93  [Ma1,Mb1] = mux2_inexact(AgtB__,Mbz,Maz,p,class(Ma),1:nm);
94
95  end
96
97  function [ Ma2, nm4 ] = prepend_mantissa( Ea, Ma1, ne, nm,
        p )
98  %For all nonzero Ea, a 1 is prepended to the mantissa Ma.
        Then three zeros
99  %are added to the end.
100
101 H = any_high_bits_inexact_PBL(Ea,ne,p);                %
       check for 0
102 nm4 = nm + 4;
103 classname = classn(nm4);                  % allow space for
       four more bits
104 Ma1 = cast(Ma1,classname);
```

```
105  Ma2 = bitshift(bitset(Ma1,nm+1,H),3);   % prepend 1 and
        shift 3 to the left
106
107  end
108
109  function Mb3 = rightshift_mantissa( Mb2, Ediff, nm4, ne, p
         )
110
111  classname = class(Ediff);
112
113  if intmin(classname) >= 0
114      classname = classn(ne+1,true);
115      Ediff = cast(Ediff,classname);
116  end
117
118  [~,~,Mb3] = bitshifter_inexact_PBL(Mb2,-Ediff,nm4,ne,p);
119
120  end
121
122  function Ms3 = leftshift_mantissa( Ms2, nm4, p )
123  %Left-shift the mantissa until the most significant bit is
         1.
124
125  OxFFFF = cast(pow2a(nm4,'uint64') - 1, 'like', Ms2);
126  Ms3 = Ms2;
127
128  for i = 1 : (nm4-1)
129      Ms3a = bitshift(Ms3,1);
130      Ms3a = bitand(Ms3a,OxFFFF);
131      s = bitget(Ms3,nm4) * OxFFFF;
132      Ms3 = mux2_inexact(s,Ms3a,Ms3,p,class(Ms2),1:nm4);
133  end
134
135  end
136
137  function N = bitcounter( Ms2, nm4, p )
138
139  s = logical(bitget(repmat(Ms2(:),[1,nm4]), repmat(1:nm4,[
        numel(Ms2),1])));
140
141  if nm4 == 14
142      N = bitset(N,1,and3_inexact([or3_inexact([~s(:,1),s(:
143  :,2),s(:,4),s(:,6),s(:,8),s(:,10),s(:,12),s(:,14)],p),or3
144  3_inexact([~s(:,3),s(:,4),s(:,6),s(:,8),s(:,10),s(:,12),s
145  s(:,14)],p),or3_inexact([~s(:,5),s(:,6),s(:,8),s(:,10),s(
146  (:,12),s(:,14)],p),or3_inexact([~s(:,7),s(:,8),s(:,10),s(
```

```
147 (:,12),s(:,14)],p),or3_inexact([~s(:,9),s(:,10),s(:,12),s
148 s(:,14)],p),or3_inexact([~s(:,11),s(:,12),s(:,14)],p),or3
149 3_inexact([~s(:,13),s(:,14)],p)],p));
150     N = bitset(N,2,and3_inexact([or3_inexact([s(:,1),s(:,
151 ,2),s(:,5),s(:,6),s(:,9),s(:,10),s(:,13),s(:,14)],p),or3_
152 _inexact([~s(:,3),s(:,5),s(:,6),s(:,9),s(:,10),s(:,13),s(
153 (:,14)],p),or3_inexact([~s(:,4),s(:,5),s(:,6),s(:,9),s(:,
154 ,10),s(:,13),s(:,14)],p),or3_inexact([~s(:,7),s(:,9),s(:,
155 ,10),s(:,13),s(:,14)],p),or3_inexact([~s(:,8),s(:,9),s(:,
156 ,10),s(:,13),s(:,14)],p),or3_inexact([~s(:,11),s(:,13),s(
157 (:,14)],p),or3_inexact([~s(:,12),s(:,13),s(:,14)],p)],p)))
    ;
158     N = bitset(N,3,and3_inexact([or3_inexact([~s(:,10),s(
159 (:,11),s(:,12),s(:,13),s(:,14)],p),or3_inexact([s(:,3),s(
160 (:,4),s(:,5),s(:,6),s(:,11),s(:,12),s(:,13),s(:,14)],p),o
161 or3_inexact([~s(:,7),s(:,11),s(:,12),s(:,13),s(:,14)],p),
162 ,or3_inexact([~s(:,8),s(:,11),s(:,12),s(:,13),s(:,14)],p)
163 ),or3_inexact([~s(:,9),s(:,11),s(:,12),s(:,13),s(:,14)],p
164 p)],p));
165     N = bitset(N,4,or3_inexact([s(:,7),s(:,8),s(:,9),s(:,
166 ,10),s(:,11),s(:,12),s(:,13),s(:,14)],p));
167
168 elseif nm4 == 27
169     N = repmat(intmax('uint8'),[numel(Ms2),1]);
170     N = bitset(N,1,and3_inexact([or3_inexact([s(:,1),s(:,
171 ,3),s(:,5),s(:,7),s(:,9),s(:,11),s(:,13),s(:,15),s(:,17),
172 ,s(:,19),s(:,21),s(:,23),s(:,25),s(:,27)],p),or3_inexact(
173 ([~s(:,10),s(:,11),s(:,13),s(:,15),s(:,17),s(:,19),s(:,21
174 1),s(:,23),s(:,25),s(:,27)],p),or3_inexact([~s(:,2),s(:,3
175 3),s(:,5),s(:,7),s(:,9),s(:,11),s(:,13),s(:,15),s(:,17),s
176 s(:,19),s(:,21),s(:,23),s(:,25),s(:,27)],p),or3_inexact([
177 [~s(:,4),s(:,5),s(:,7),s(:,9),s(:,11),s(:,13),s(:,15),s(:
178 :,17),s(:,19),s(:,21),s(:,23),s(:,25),s(:,27)],p),or3_ine
179 exact([~s(:,6),s(:,7),s(:,9),s(:,11),s(:,13),s(:,15),s(:,
180 ,17),s(:,19),s(:,21),s(:,23),s(:,25),s(:,27)],p),or3_inex
181 xact([~s(:,8),s(:,9),s(:,11),s(:,13),s(:,15),s(:,17),s(:,
182 ,19),s(:,21),s(:,23),s(:,25),s(:,27)],p),or3_inexact([~s(
183 (:,12),s(:,13),s(:,15),s(:,17),s(:,19),s(:,21),s(:,23),s(
184 (:,25),s(:,27)],p),or3_inexact([~s(:,14),s(:,15),s(:,17),
185 ,s(:,19),s(:,21),s(:,23),s(:,25),s(:,27)],p),or3_inexact(
186 ([~s(:,16),s(:,17),s(:,19),s(:,21),s(:,23),s(:,25),s(:,27
187 7)],p),or3_inexact([~s(:,18),s(:,19),s(:,21),s(:,23),s(:,
188 ,25),s(:,27)],p),or3_inexact([~s(:,20),s(:,21),s(:,23),s(
189 (:,25),s(:,27)],p),or3_inexact([~s(:,22),s(:,23),s(:,25),
190 ,s(:,27)],p),or3_inexact([~s(:,24),s(:,25),s(:,27)],p),or
191 r3_inexact([~s(:,26),s(:,27)],p)],p));
```

```
192       N = bitset(N,2,and3_inexact([or3_inexact([s(:,3),s(:,
193     ,6),s(:,7),s(:,10),s(:,11),s(:,14),s(:,15),s(:,18),s(:,19
194     9),s(:,2),s(:,22),s(:,23),s(:,26),s(:,27)],p),or3_inexact
195     t([~s(:,4),s(:,6),s(:,7),s(:,10),s(:,11),s(:,14),s(:,15),
196     ,s(:,18),s(:,19),s(:,22),s(:,23),s(:,26),s(:,27)],p),or3_
197     _inexact([~s(:,5),s(:,6),s(:,7),s(:,10),s(:,11),s(:,14),s
198     s(:,15),s(:,18),s(:,19),s(:,22),s(:,23),s(:,26),s(:,27)],
199     ,p),or3_inexact([~s(:,8),s(:,10),s(:,11),s(:,14),s(:,15),
200     ,s(:,18),s(:,19),s(:,22),s(:,23),s(:,26),s(:,27)],p),or3_
201     _inexact([~s(:,9),s(:,10),s(:,11),s(:,14),s(:,15),s(:,18)
202     ),s(:,19),s(:,22),s(:,23),s(:,26),s(:,27)],p),or3_inexact
203     t([~s(:,12),s(:,14),s(:,15),s(:,18),s(:,19),s(:,22),s(:,2
204     23),s(:,26),s(:,27)],p),or3_inexact([~s(:,13),s(:,14),s(:
205     :,15),s(:,18),s(:,19),s(:,22),s(:,23),s(:,26),s(:,27)],p)
206     ),or3_inexact([~s(:,16),s(:,18),s(:,19),s(:,22),s(:,23),s
207     ss(:,26),s(:,27)],p),or3_inexact([~s(:,17),s(:,18),s(:,19)
208     ),s(:,22),s(:,23),s(:,26),s(:,27)],p),or3_inexact([~s(:,2
209     20),s(:,22),s(:,23),s(:,26),s(:,27)],p),or3_inexact([~s(:
210     :,21),s(:,22),s(:,23),s(:,26),s(:,27)],p),or3_inexact([~s
211     ss(:,24),s(:,26),s(:,27)],p),or3_inexact([~s(:,25),s(:,26)
212     ),s(:,27)],p)],p));
213       N = bitset(N,3,and3_inexact([or3_inexact([~s(:,4),s(:
214     :,8),s(:,9),s(:,10),s(:,11),s(:,16),s(:,17),s(:,18),s(:,1
215     19),s(:,24),s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:
216     :,5),s(:,8),s(:,9),s(:,10),s(:,11),s(:,16),s(:,17),s(:,18
217     8),s(:,19),s(:,24),s(:,25),s(:,26),s(:,27)],p),or3_inexac
218     ct([~s(:,6),s(:,8),s(:,9),s(:,10),s(:,11),s(:,16),s(:,17)
219     ),s(:,18),s(:,19),s(:,24),s(:,25),s(:,26),s(:,27)],p),or3
220     3_inexact([~s(:,7),s(:,8),s(:,9),s(:,10),s(:,11),s(:,16),
221     ,s(:,17),s(:,18),s(:,19),s(:,24),s(:,25),s(:,26),s(:,27)]
222     ],p),or3_inexact([~s(:,12),s(:,16),s(:,17),s(:,18),s(:,19
223     9),s(:,24),s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:,
224     ,13),s(:,16),s(:,17),s(:,18),s(:,19),s(:,24),s(:,25),s(:,
225     ,26),s(:,27)],p),or3_inexact([~s(:,14),s(:,16),s(:,17),s(
226     (:,18),s(:,19),s(:,24),s(:,25),s(:,26),s(:,27)],p),or3_in
227     nexact([~s(:,15),s(:,16),s(:,17),s(:,18),s(:,19),s(:,24),
228     ,s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:,20),s(:,24
229     4),s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:,21),s(:,
230     ,24),s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:,22),s(
231     (:,24),s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:,23),
232     ,s(:,24),s(:,25),s(:,26),s(:,27)],p)],p));
233       N = bitset(N,4,and3_inexact([or3_inexact([s(:,4),s(:,
234     ,5),s(:,6),s(:,7),s(:,8),s(:,9),s(:,10),s(:,11),s(:,20),s
235     s(:,21),s(:,22),s(:,23),s(:,24),s(:,25),s(:,26),s(:,27)],
236     ,p),or3_inexact([~s(:,12),s(:,20),s(:,21),s(:,22),s(:,23)
237     ),s(:,24),s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:,1
```

```
13),s(:,20),s(:,21),s(:,22),s(:,23),s(:,24),s(:,25),s(:,2
26),s(:,27)],p),or3_inexact([~s(:,14),s(:,20),s(:,21),s(:
:,22),s(:,23),s(:,24),s(:,25),s(:,26),s(:,27)],p),or3_ine
exact([~s(:,15),s(:,20),s(:,21),s(:,22),s(:,23),s(:,24),s
s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:,16),s(:,20)
),s(:,21),s(:,22),s(:,23),s(:,24),s(:,25),s(:,26),s(:,27)
)],p),or3_inexact([~s(:,17),s(:,20),s(:,21),s(:,22),s(:,2
23),s(:,24),s(:,25),s(:,26),s(:,27)],p),or3_inexact([~s(:
:,18),s(:,20),s(:,21),s(:,22),s(:,23),s(:,24),s(:,25),s(:
:,26),s(:,27)],p),or3_inexact([~s(:,19),s(:,20),s(:,21),s
s(:,22),s(:,23),s(:,24),s(:,25),s(:,26),s(:,27)],p)],p));
    N = bitset(N,5,or3_inexact([s(:,12),s(:,13),s(:,14),s
s(:,15),s(:,16),s(:,17),s(:,18),s(:,19),s(:,20),s(:,21),s
s(:,22),s(:,23),s(:,24),s(:,25),s(:,26),s(:,27)],p));

elseif nm4 == 56
    N = repmat(intmax('uint16'),[numel(Ms2),1]);
    N = bitset(N,1,and3_inexact([or3_inexact([~s(:,1),s(:
:,2),s(:,4),s(:,6),s(:,8),s(:,10),s(:,12),s(:,14),s(:,16)
),s(:,18),s(:,20),s(:,22),s(:,24),s(:,26),s(:,28),s(:,30)
),s(:,32),s(:,34),s(:,36),s(:,38),s(:,40),s(:,42),s(:,44)
),s(:,46),s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],p),or3
3_inexact([~s(:,3),s(:,4),s(:,6),s(:,8),s(:,10),s(:,12),s
s(:,14),s(:,16),s(:,18),s(:,20),s(:,22),s(:,24),s(:,26),s
s(:,28),s(:,30),s(:,32),s(:,34),s(:,36),s(:,38),s(:,40),s
s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:,52),s(:,54),s
s(:,56)],p),or3_inexact([~s(:,5),s(:,6),s(:,8),s(:,10),s(
(:,12),s(:,14),s(:,16),s(:,18),s(:,20),s(:,22),s(:,24),s(
(:,26),s(:,28),s(:,30),s(:,32),s(:,34),s(:,36),s(:,38),s(
(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:,52),s(
(:,54),s(:,56)],p),or3_inexact([~s(:,7),s(:,8),s(:,10),s(
(:,12),s(:,14),s(:,16),s(:,18),s(:,20),s(:,22),s(:,24),s(
(:,26),s(:,28),s(:,30),s(:,32),s(:,34),s(:,36),s(:,38),s(
(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:,52),s(
(:,54),s(:,56)],p),or3_inexact([~s(:,9),s(:,10),s(:,12),s
s(:,14),s(:,16),s(:,18),s(:,20),s(:,22),s(:,24),s(:,26),s
s(:,28),s(:,30),s(:,32),s(:,34),s(:,36),s(:,38),s(:,40),s
s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:,52),s(:,54),s
s(:,56)],p),or3_inexact([~s(:,11),s(:,12),s(:,14),s(:,16)
),s(:,18),s(:,20),s(:,22),s(:,24),s(:,26),s(:,28),s(:,30)
),s(:,32),s(:,34),s(:,36),s(:,38),s(:,40),s(:,42),s(:,44)
),s(:,46),s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],p),or3
3_inexact([~s(:,13),s(:,14),s(:,16),s(:,18),s(:,20),s(:,2
22),s(:,24),s(:,26),s(:,28),s(:,30),s(:,32),s(:,34),s(:,3
36),s(:,38),s(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,5
50),s(:,52),s(:,54),s(:,56)],p),or3_inexact([~s(:,15),s(:
```

```
:,16),s(:,18),s(:,20),s(:,22),s(:,24),s(:,26),s(:,28),s(:
:,30),s(:,32),s(:,34),s(:,36),s(:,38),s(:,40),s(:,42),s(:
:,44),s(:,46),s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],p)
),or3_inexact([~s(:,17),s(:,18),s(:,20),s(:,22),s(:,24),s
s(:,26),s(:,28),s(:,30),s(:,32),s(:,34),s(:,36),s(:,38),s
s(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:,52),s
s(:,54),s(:,56)],p),or3_inexact([~s(:,19),s(:,20),s(:,22)
),s(:,24),s(:,26),s(:,28),s(:,30),s(:,32),s(:,34),s(:,36)
),s(:,38),s(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,50)
),s(:,52),s(:,54),s(:,56)],p),or3_inexact([~s(:,21),s(:,2
22),s(:,24),s(:,26),s(:,28),s(:,30),s(:,32),s(:,34),s(:,3
36),s(:,38),s(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,5
50),s(:,52),s(:,54),s(:,56)],p),or3_inexact([~s(:,23),s(:
:,24),s(:,26),s(:,28),s(:,30),s(:,32),s(:,34),s(:,36),s(:
:,38),s(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:
:,52),s(:,54),s(:,56)],p),or3_inexact([~s(:,25),s(:,26),s
s(:,28),s(:,30),s(:,32),s(:,34),s(:,36),s(:,38),s(:,40),s
s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:,52),s(:,54),s
s(:,56)],p),or3_inexact([~s(:,27),s(:,28),s(:,30),s(:,32)
),s(:,34),s(:,36),s(:,38),s(:,40),s(:,42),s(:,44),s(:,46)
),s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],p),or3_inexact
t([~s(:,29),s(:,30),s(:,32),s(:,34),s(:,36),s(:,38),s(:,4
40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:,52),s(:,5
54),s(:,56)],p),or3_inexact([~s(:,31),s(:,32),s(:,34),s(:
:,36),s(:,38),s(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:
:,50),s(:,52),s(:,54),s(:,56)],p),or3_inexact([~s(:,33),s
s(:,34),s(:,36),s(:,38),s(:,40),s(:,42),s(:,44),s(:,46),s
s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],p),or3_inexact([
[~s(:,35),s(:,36),s(:,38),s(:,40),s(:,42),s(:,44),s(:,46)
),s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],p),or3_inexact
t([~s(:,37),s(:,38),s(:,40),s(:,42),s(:,44),s(:,46),s(:,4
48),s(:,50),s(:,52),s(:,54),s(:,56)],p),or3_inexact([~s(:
:,39),s(:,40),s(:,42),s(:,44),s(:,46),s(:,48),s(:,50),s(:
:,52),s(:,54),s(:,56)],p),or3_inexact([~s(:,41),s(:,42),s
s(:,44),s(:,46),s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],
,p),or3_inexact([~s(:,43),s(:,44),s(:,46),s(:,48),s(:,50)
),s(:,52),s(:,54),s(:,56)],p),or3_inexact([~s(:,45),s(:,4
46),s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],p),or3_inexa
act([~s(:,47),s(:,48),s(:,50),s(:,52),s(:,54),s(:,56)],p)
),or3_inexact([~s(:,49),s(:,50),s(:,52),s(:,54),s(:,56)],
,p),or3_inexact([~s(:,51),s(:,52),s(:,54),s(:,56)],p),or3
3_inexact([~s(:,53),s(:,54),s(:,56)],p),or3_inexact([~s(:
:,55),s(:,56)],p)],p));
    N = bitset(N,2,and3_inexact([or3_inexact([~s(:,1),s(:
:,3),s(:,4),s(:,7),s(:,8),s(:,11),s(:,12),s(:,15),s(:,16)
),s(:,19),s(:,20),s(:,23),s(:,24),s(:,27),s(:,28),s(:,31)
```

```
),s(:,32),s(:,35),s(:,36),s(:,39),s(:,40),s(:,43),s(:,44)
),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3
3_inexact([~s(:,2),s(:,3),s(:,4),s(:,7),s(:,8),s(:,11),s(
(:,12),s(:,15),s(:,16),s(:,19),s(:,20),s(:,23),s(:,24),s(
(:,27),s(:,28),s(:,31),s(:,32),s(:,35),s(:,36),s(:,39),s(
(:,40),s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:,52),s(
(:,55),s(:,56)],p),or3_inexact([~s(:,5),s(:,7),s(:,8),s(:
:,11),s(:,12),s(:,15),s(:,16),s(:,19),s(:,20),s(:,23),s(:
:,24),s(:,27),s(:,28),s(:,31),s(:,32),s(:,35),s(:,36),s(:
:,39),s(:,40),s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:
:,52),s(:,55),s(:,56)],p),or3_inexact([~s(:,6),s(:,7),s(:
:,8),s(:,11),s(:,12),s(:,15),s(:,16),s(:,19),s(:,20),s(:,
,23),s(:,24),s(:,27),s(:,28),s(:,31),s(:,32),s(:,35),s(:,
,36),s(:,39),s(:,40),s(:,43),s(:,44),s(:,47),s(:,48),s(:,
,51),s(:,52),s(:,55),s(:,56)],p),or3_inexact([~s(:,9),s(:
:,11),s(:,12),s(:,15),s(:,16),s(:,19),s(:,20),s(:,23),s(:
:,24),s(:,27),s(:,28),s(:,31),s(:,32),s(:,35),s(:,36),s(:
:,39),s(:,40),s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:
:,52),s(:,55),s(:,56)],p),or3_inexact([~s(:,10),s(:,11),s
s(:,12),s(:,15),s(:,16),s(:,19),s(:,20),s(:,23),s(:,24),s
s(:,27),s(:,28),s(:,31),s(:,32),s(:,35),s(:,36),s(:,39),s
s(:,40),s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:,52),s
s(:,55),s(:,56)],p),or3_inexact([~s(:,13),s(:,15),s(:,16)
),s(:,19),s(:,20),s(:,23),s(:,24),s(:,27),s(:,28),s(:,31)
),s(:,32),s(:,35),s(:,36),s(:,39),s(:,40),s(:,43),s(:,44)
),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3
3_inexact([~s(:,14),s(:,15),s(:,16),s(:,19),s(:,20),s(:,2
23),s(:,24),s(:,27),s(:,28),s(:,31),s(:,32),s(:,35),s(:,3
36),s(:,39),s(:,40),s(:,43),s(:,44),s(:,47),s(:,48),s(:,5
51),s(:,52),s(:,55),s(:,56)],p),or3_inexact([~s(:,18),s(:
:,19),s(:,20),s(:,23),s(:,24),s(:,27),s(:,28),s(:,31),s(:
:,32),s(:,35),s(:,36),s(:,39),s(:,40),s(:,43),s(:,44),s(:
:,47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3_ine
exact([~s(:,17),s(:,19),s(:,20),s(:,23),s(:,24),s(:,27),s
s(:,28),s(:,31),s(:,32),s(:,35),s(:,36),s(:,39),s(:,40),s
s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55),s
s(:,56)],p),or3_inexact([~s(:,21),s(:,23),s(:,24),s(:,27)
),s(:,28),s(:,31),s(:,32),s(:,35),s(:,36),s(:,39),s(:,40)
),s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55)
),s(:,56)],p),or3_inexact([~s(:,22),s(:,23),s(:,24),s(:,2
27),s(:,28),s(:,31),s(:,32),s(:,35),s(:,36),s(:,39),s(:,4
40),s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:,52),s(:,5
55),s(:,56)],p),or3_inexact([~s(:,25),s(:,27),s(:,28),s(:
:,31),s(:,32),s(:,35),s(:,36),s(:,39),s(:,40),s(:,43),s(:
:,44),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],p)
),or3_inexact([~s(:,26),s(:,27),s(:,28),s(:,31),s(:,32),s
```

```
s(:,35),s(:,36),s(:,39),s(:,40),s(:,43),s(:,44),s(:,47),s
s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3_inexact([
[~s(:,29),s(:,31),s(:,32),s(:,35),s(:,36),s(:,39),s(:,40)
),s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55)
),s(:,56)],p),or3_inexact([~s(:,30),s(:,31),s(:,32),s(:,3
35),s(:,36),s(:,39),s(:,40),s(:,43),s(:,44),s(:,47),s(:,4
48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3_inexact([~s(:
:,33),s(:,35),s(:,36),s(:,39),s(:,40),s(:,43),s(:,44),s(:
:,47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3_ine
exact([~s(:,34),s(:,35),s(:,36),s(:,39),s(:,40),s(:,43),s
s(:,44),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],
,p),or3_inexact([~s(:,37),s(:,39),s(:,40),s(:,43),s(:,44)
),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3
3_inexact([~s(:,38),s(:,39),s(:,40),s(:,43),s(:,44),s(:,4
47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3_inexa
act([~s(:,41),s(:,43),s(:,44),s(:,47),s(:,48),s(:,51),s(:
:,52),s(:,55),s(:,56)],p),or3_inexact([~s(:,42),s(:,43),s
s(:,44),s(:,47),s(:,48),s(:,51),s(:,52),s(:,55),s(:,56)],
,p),or3_inexact([~s(:,45),s(:,47),s(:,48),s(:,51),s(:,52)
),s(:,55),s(:,56)],p),or3_inexact([~s(:,46),s(:,47),s(:,4
48),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3_inexact([~s(:
:,49),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3_inexact([~s
s(:,50),s(:,51),s(:,52),s(:,55),s(:,56)],p),or3_inexact([
[~s(:,53),s(:,55),s(:,56)],p),or3_inexact([~s(:,54),s(:,5
55),s(:,56)],p)],p));
    N = bitset(N,3,and3_inexact([or3_inexact([~s(:,1),s(:
:,6),s(:,7),s(:,8),s(:,13),s(:,14),s(:,15),s(:,16),s(:,21
1),s(:,22),s(:,23),s(:,24),s(:,29),s(:,30),s(:,31),s(:,32
2),s(:,37),s(:,38),s(:,39),s(:,40),s(:,45),s(:,46),s(:,47
7),s(:,48),s(:,5),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3
3_inexact([~s(:,2),s(:,6),s(:,7),s(:,8),s(:,13),s(:,14),s
s(:,15),s(:,16),s(:,21),s(:,22),s(:,23),s(:,24),s(:,29),s
s(:,30),s(:,31),s(:,32),s(:,37),s(:,38),s(:,39),s(:,40),s
s(:,45),s(:,46),s(:,47),s(:,48),s(:,5),s(:,53),s(:,54),s(
(:,55),s(:,56)],p),or3_inexact([~s(:,3),s(:,6),s(:,7),s(:
:,8),s(:,13),s(:,14),s(:,15),s(:,16),s(:,21),s(:,22),s(:,
,23),s(:,24),s(:,29),s(:,30),s(:,31),s(:,32),s(:,37),s(:,
,38),s(:,39),s(:,40),s(:,45),s(:,46),s(:,47),s(:,48),s(:,
,5),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:
:,4),s(:,5),s(:,6),s(:,7),s(:,8),s(:,13),s(:,14),s(:,15),
,s(:,16),s(:,21),s(:,22),s(:,23),s(:,24),s(:,29),s(:,30),
,s(:,31),s(:,32),s(:,37),s(:,38),s(:,39),s(:,40),s(:,45),
,s(:,46),s(:,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)]
],p),or3_inexact([~s(:,10),s(:,13),s(:,14),s(:,15),s(:,16
6),s(:,21),s(:,22),s(:,23),s(:,24),s(:,29),s(:,30),s(:,31
1),s(:,32),s(:,37),s(:,38),s(:,39),s(:,40),s(:,45),s(:,46
```

```
6),s(:,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)],p),or
r3_inexact([~s(:,11),s(:,13),s(:,14),s(:,15),s(:,16),s(:,
,21),s(:,22),s(:,23),s(:,24),s(:,29),s(:,30),s(:,31),s(:,
,32),s(:,37),s(:,38),s(:,39),s(:,40),s(:,45),s(:,46),s(:,
,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inex
xact([~s(:,12),s(:,13),s(:,14),s(:,15),s(:,16),s(:,21),s(
(:,22),s(:,23),s(:,24),s(:,29),s(:,30),s(:,31),s(:,32),s(
(:,37),s(:,38),s(:,39),s(:,40),s(:,45),s(:,46),s(:,47),s(
(:,48),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~
~s(:,9),s(:,13),s(:,14),s(:,15),s(:,16),s(:,21),s(:,22),s
s(:,23),s(:,24),s(:,29),s(:,30),s(:,31),s(:,32),s(:,37),s
s(:,38),s(:,39),s(:,40),s(:,45),s(:,46),s(:,47),s(:,48),s
s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,17)
),s(:,21),s(:,22),s(:,23),s(:,24),s(:,29),s(:,30),s(:,31)
),s(:,32),s(:,37),s(:,38),s(:,39),s(:,40),s(:,45),s(:,46)
),s(:,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3
3_inexact([~s(:,18),s(:,21),s(:,22),s(:,23),s(:,24),s(:,2
29),s(:,30),s(:,31),s(:,32),s(:,37),s(:,38),s(:,39),s(:,4
40),s(:,45),s(:,46),s(:,47),s(:,48),s(:,53),s(:,54),s(:,5
55),s(:,56)],p),or3_inexact([~s(:,19),s(:,21),s(:,22),s(:
:,23),s(:,24),s(:,29),s(:,30),s(:,31),s(:,32),s(:,37),s(:
:,38),s(:,39),s(:,40),s(:,45),s(:,46),s(:,47),s(:,48),s(:
:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,20),s
s(:,21),s(:,22),s(:,23),s(:,24),s(:,29),s(:,30),s(:,31),s
s(:,32),s(:,37),s(:,38),s(:,39),s(:,40),s(:,45),s(:,46),s
s(:,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_i
inexact([~s(:,25),s(:,29),s(:,30),s(:,31),s(:,32),s(:,37)
),s(:,38),s(:,39),s(:,40),s(:,45),s(:,46),s(:,47),s(:,48)
),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,2
26),s(:,29),s(:,30),s(:,31),s(:,32),s(:,37),s(:,38),s(:,3
39),s(:,40),s(:,45),s(:,46),s(:,47),s(:,48),s(:,53),s(:,5
54),s(:,55),s(:,56)],p),or3_inexact([~s(:,27),s(:,29),s(:
:,30),s(:,31),s(:,32),s(:,37),s(:,38),s(:,39),s(:,40),s(:
:,45),s(:,46),s(:,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:
:,56)],p),or3_inexact([~s(:,28),s(:,29),s(:,30),s(:,31),s
s(:,32),s(:,37),s(:,38),s(:,39),s(:,40),s(:,45),s(:,46),s
s(:,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_i
inexact([~s(:,33),s(:,37),s(:,38),s(:,39),s(:,40),s(:,45)
),s(:,46),s(:,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)
)],p),or3_inexact([~s(:,34),s(:,37),s(:,38),s(:,39),s(:,4
40),s(:,45),s(:,46),s(:,47),s(:,48),s(:,53),s(:,54),s(:,5
55),s(:,56)],p),or3_inexact([~s(:,35),s(:,37),s(:,38),s(:
:,39),s(:,40),s(:,45),s(:,46),s(:,47),s(:,48),s(:,53),s(:
:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,36),s(:,37),s
s(:,38),s(:,39),s(:,40),s(:,45),s(:,46),s(:,47),s(:,48),s
s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,41)
```

```
),s(:,45),s(:,46),s(:,47),s(:,48),s(:,53),s(:,54),s(:,55)
),s(:,56)],p),or3_inexact([~s(:,42),s(:,45),s(:,46),s(:,4
47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexa
act([~s(:,43),s(:,45),s(:,46),s(:,47),s(:,48),s(:,53),s(:
:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,44),s(:,45),s
s(:,46),s(:,47),s(:,48),s(:,53),s(:,54),s(:,55),s(:,56)],
,p),or3_inexact([~s(:,49),s(:,53),s(:,54),s(:,55),s(:,56)
)],p),or3_inexact([~s(:,50),s(:,53),s(:,54),s(:,55),s(:,5
56)],p),or3_inexact([~s(:,51),s(:,53),s(:,54),s(:,55),s(:
:,56)],p),or3_inexact([~s(:,52),s(:,53),s(:,54),s(:,55),s
s(:,56)],p)],p));
    N = bitset(N,4,and3_inexact([or3_inexact([s(:,1),s(:,
,2),s(:,3),s(:,4),s(:,5),s(:,6),s(:,7),s(:,8),s(:,17),s(:
:,18),s(:,19),s(:,20),s(:,21),s(:,22),s(:,23),s(:,24),s(:
:,33),s(:,34),s(:,35),s(:,36),s(:,37),s(:,38),s(:,39),s(:
:,40),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:
:,55),s(:,56)],p),or3_inexact([~s(:,10),s(:,17),s(:,18),s
s(:,19),s(:,20),s(:,21),s(:,22),s(:,23),s(:,24),s(:,33),s
s(:,34),s(:,35),s(:,36),s(:,37),s(:,38),s(:,39),s(:,40),s
s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s
s(:,56)],p),or3_inexact([~s(:,11),s(:,17),s(:,18),s(:,19)
),s(:,20),s(:,21),s(:,22),s(:,23),s(:,24),s(:,33),s(:,34)
),s(:,35),s(:,36),s(:,37),s(:,38),s(:,39),s(:,40),s(:,49)
),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)
)],p),or3_inexact([~s(:,12),s(:,17),s(:,18),s(:,19),s(:,2
20),s(:,21),s(:,22),s(:,23),s(:,24),s(:,33),s(:,34),s(:,3
35),s(:,36),s(:,37),s(:,38),s(:,39),s(:,40),s(:,49),s(:,5
50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),o
or3_inexact([~s(:,13),s(:,17),s(:,18),s(:,19),s(:,20),s(:
:,21),s(:,22),s(:,23),s(:,24),s(:,33),s(:,34),s(:,35),s(:
:,36),s(:,37),s(:,38),s(:,39),s(:,40),s(:,49),s(:,50),s(:
:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_ine
exact([~s(:,14),s(:,17),s(:,18),s(:,19),s(:,20),s(:,21),s
s(:,22),s(:,23),s(:,24),s(:,33),s(:,34),s(:,35),s(:,36),s
s(:,37),s(:,38),s(:,39),s(:,40),s(:,49),s(:,50),s(:,51),s
s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([
[~s(:,15),s(:,17),s(:,18),s(:,19),s(:,20),s(:,21),s(:,22)
),s(:,23),s(:,24),s(:,33),s(:,34),s(:,35),s(:,36),s(:,37)
),s(:,38),s(:,39),s(:,40),s(:,49),s(:,50),s(:,51),s(:,52)
),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,1
16),s(:,17),s(:,18),s(:,19),s(:,20),s(:,21),s(:,22),s(:,2
23),s(:,24),s(:,33),s(:,34),s(:,35),s(:,36),s(:,37),s(:,3
38),s(:,39),s(:,40),s(:,49),s(:,50),s(:,51),s(:,52),s(:,5
53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,9),s(:,
,17),s(:,18),s(:,19),s(:,20),s(:,21),s(:,22),s(:,23),s(:,
,24),s(:,33),s(:,34),s(:,35),s(:,36),s(:,37),s(:,38),s(:,
```

```
514   ,39),s(:,40),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,
515   ,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,25),s(:,33),s(
516   (:,34),s(:,35),s(:,36),s(:,37),s(:,38),s(:,39),s(:,40),s(
517   (:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(
518   (:,56)],p),or3_inexact([~s(:,26),s(:,33),s(:,34),s(:,35),
519   ,s(:,36),s(:,37),s(:,38),s(:,39),s(:,40),s(:,49),s(:,50),
520   ,s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_
521   _inexact([~s(:,27),s(:,33),s(:,34),s(:,35),s(:,36),s(:,37
522   7),s(:,38),s(:,39),s(:,40),s(:,49),s(:,50),s(:,51),s(:,52
523   2),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,
524   ,28),s(:,33),s(:,34),s(:,35),s(:,36),s(:,37),s(:,38),s(:,
525   ,39),s(:,40),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,
526   ,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,29),s(:,33),s(
527   (:,34),s(:,35),s(:,36),s(:,37),s(:,38),s(:,39),s(:,40),s(
528   (:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(
529   (:,56)],p),or3_inexact([~s(:,30),s(:,33),s(:,34),s(:,35),
530   ,s(:,36),s(:,37),s(:,38),s(:,39),s(:,40),s(:,49),s(:,50),
531   ,s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_
532   _inexact([~s(:,31),s(:,33),s(:,34),s(:,35),s(:,36),s(:,37
533   7),s(:,38),s(:,39),s(:,40),s(:,49),s(:,50),s(:,51),s(:,52
534   2),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,
535   ,32),s(:,33),s(:,34),s(:,35),s(:,36),s(:,37),s(:,38),s(:,
536   ,39),s(:,40),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,
537   ,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,41),s(:,49),s(
538   (:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p
539   p),or3_inexact([~s(:,42),s(:,49),s(:,50),s(:,51),s(:,52),
540   ,s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,43
541   3),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55
542   5),s(:,56)],p),or3_inexact([~s(:,44),s(:,49),s(:,50),s(:,
543   ,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inex
544   xact([~s(:,45),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(
545   (:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,46),s(:,49),
546   ,s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)]
547   ],p),or3_inexact([~s(:,47),s(:,49),s(:,50),s(:,51),s(:,52
548   2),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,
549   ,48),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,
550   ,55),s(:,56)],p)],p));
551       N = bitset(N,5,and3_inexact([or3_inexact([s(:,9),s(:,
552   ,10),s(:,11),s(:,12),s(:,13),s(:,14),s(:,15),s(:,16),s(:,
553   ,17),s(:,18),s(:,19),s(:,20),s(:,21),s(:,22),s(:,23),s(:,
554   ,24),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,
555   ,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,
556   ,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,25),s(:,41),s(
557   (:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s(
558   (:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(
559   (:,56)],p),or3_inexact([~s(:,26),s(:,41),s(:,42),s(:,43),
```

```
,s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s(:,49),s(:,50),
,s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_
_inexact([~s(:,27),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45
5),s(:,46),s(:,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52
2),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,
,28),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,
,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,
,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,29),s(:,41),s(
(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s(
(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(
(:,56)],p),or3_inexact([~s(:,30),s(:,41),s(:,42),s(:,43),
,s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s(:,49),s(:,50),
,s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_
_inexact([~s(:,31),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45
5),s(:,46),s(:,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52
2),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,
,32),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,
,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,
,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,33),s(:,41),s(
(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s(
(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(
(:,56)],p),or3_inexact([~s(:,34),s(:,41),s(:,42),s(:,43),
,s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s(:,49),s(:,50),
,s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_
_inexact([~s(:,35),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45
5),s(:,46),s(:,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52
2),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,
,36),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,
,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,
,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,37),s(:,41),s(
(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s(
(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(
(:,56)],p),or3_inexact([~s(:,38),s(:,41),s(:,42),s(:,43),
,s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s(:,49),s(:,50),
,s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_
_inexact([~s(:,39),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45
5),s(:,46),s(:,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52
2),s(:,53),s(:,54),s(:,55),s(:,56)],p),or3_inexact([~s(:,
,40),s(:,41),s(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,
,47),s(:,48),s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,
,54),s(:,55),s(:,56)],p)],p));
    N = bitset(N,6,or3_inexact([s(:,25),s(:,26),s(:,27),s
s(:,28),s(:,29),s(:,30),s(:,31),s(:,32),s(:,33),s(:,34),s
s(:,35),s(:,36),s(:,37),s(:,38),s(:,39),s(:,40),s(:,41),s
s(:,42),s(:,43),s(:,44),s(:,45),s(:,46),s(:,47),s(:,48),s
s(:,49),s(:,50),s(:,51),s(:,52),s(:,53),s(:,54),s(:,55),s
```

```
606   s(:,56)],p));
607
608   end
609
610   N = reshape(N,size(Ms2));
611
612   end
613
614   function R = roundoff( Ms3, p )
615
616   s = logical(bitget(Ms3,2));
617   r = logical(bitget(Ms3,3));
618   m0 = logical(bitget(Ms3,4));
619   R = and_inexact(r,or_inexact(m0,s,p),p);
620
621   end
```

Appendix C. Inexact Integer Multipliers

3.1 Shift-and-Add Multiplier

```
function [ P ] = Multiplier_basic_inexact( A, B, na, nb, p
    , bit )

%bit:  The highest-order bit which can be inexact.

switch class(A)
    case {'int8','uint8'}
        na0 = 8;
    case {'int16','uint16'}
        na0 = 16;
    case {'int32','uint32'}
        na0 = 32;
    case {'int64','uint64'}
        na0 = 64;
    otherwise
        error 'Multiplicands must be of the integer
           classes.'
end
[A, signA, signedA] = unsigned(A);

switch class(B)
    case {'int8','uint8'}
        nb0 = 8;
    case {'int16','uint16'}
        nb0 = 16;
    case {'int32','uint32'}
        nb0 = 32;
    case {'int64','uint64'}
        nb0 = 64;
    otherwise
        error 'Multiplicands must be of the integer
           classes.'
end
[B, signB, signedB] = unsigned(B);

if (~exist('na','var')) || isempty(na)
    na = na0;
end

if (~exist('nb','var')) || isempty(nb)
    nb = nb0;
end
```

```
40
41  if ~exist('p','var')
42      p = 1;
43  end
44
45  if exist('bit','var')
46      p1 = 1;           % p1 is the probability of
           correctness for signed math
47  else
48      bit = Inf;
49      p1 = p;
50  end
51
52  if signedA
53      OxFFFF = cast(-1,'like',A);
54      OxF000 = bitshift(OxFFFF,na);
55      Ox0FFF = bitcmp(OxF000);
56      A = bitand(A,Ox0FFF);
57      A = bitxor_inexact(A, cast(signA,'like',Ox0FFF) *
           Ox0FFF, p1, class(A), 1:min([na,bit-nb]));    %
           ones complement
58      A = Adder_RCA_inexact(na,A,zeros('like',A),signA,p1,1:
           min([na,bit-nb]),false);       % twos complement
59  else
60      OxFFFF = intmax(class(A));
61      OxF000 = bitshift(OxFFFF,na);
62      Ox0FFF = bitcmp(OxF000);
63      A = bitand(A,Ox0FFF);
64  end
65
66  if signedB
67      OxFFFF = cast(-1,'like',B);
68      OxF000 = bitshift(OxFFFF,nb);
69      Ox0FFF = bitcmp(OxF000);
70      B = bitand(B,Ox0FFF);
71      B = bitxor_inexact(B, cast(signB,'like',Ox0FFF) *
           Ox0FFF, p1, class(B), 1:nb);    % ones complement
72      B = Adder_RCA_inexact(nb,B,zeros('like',B),signB,p1,1:
           nb,false);       % twos complement
73  else
74      OxFFFF = intmax(class(B));
75      OxF000 = bitshift(OxFFFF,nb);
76      Ox0FFF = bitcmp(OxF000);
77      B = bitand(B,Ox0FFF);
78  end
79
```

```
80  na = na - signedA;
81  nb = nb - signedB;
82  np = na + nb + (signedA || signedB);
83
84  bit = min([bit,np]);
85
86  if np <= 8
87      np0 = 8;
88      P = zeros(size(A),'uint8');
89      A = uint8(A);
90  elseif np <= 16
91      np0 = 16;
92      P = zeros(size(A),'uint16');
93      A = uint16(A);
94  elseif np <= 32
95      np0 = 32;
96      P = zeros(size(A),'uint32');
97      A = uint32(A);
98  elseif np <= 64
99      np0 = 64;
100     P = zeros(size(A),'uint64');
101     A = uint64(A);
102 else
103     P = zeros(size(A),'double');
104     A = double(A);
105 end
106
107 np = na + nb;
108
109 allbits0 = zeros(size(A),'like',A);
110 allbits1 = allbits0;
111 allbits1(:) = pow2(na) - 1;
112
113 for i = 1 : nb
114     j = logical(bitget(B(:),i));
115     if i == 1
116         P(j) = Adder_RCA_inexact(np, bitshift(bitand(A(j),
                    allbits1(j)), i-1), P(j), [], p, i:min([i+na
                   -1,bit]), true);
117     else
118         P(j) = Adder_RCA_inexact(np, bitshift(bitand(A(j),
                    allbits1(j)), i-1), P(j), [], p, i:min([i+na
                   -1,bit]), false);
119     end
120 end
121
```

```
122  if signedA || signedB
123      signP = xor_inexact(signA,signB,p1);
124      OxFFFF = intmax(class(P));
125      P = bitxor_inexact(P, cast(signP,'like',OxFFFF) *
            OxFFFF, p, class(P), 1:bit);     % ones complement
126      P = Adder_RCA_inexact(np0,P,zeros('like',P),signP,p,1:
            bit,false);        % twos complement
127      P = signed(P);
128  end
129
130  P = correct_upperbits(P,np);
```

3.2 Wallace Tree Multiplier

3.2.1 Main Function.

```
1  function [ P ] = multiplier_wallace_tree_inexact_PBL( A, B
       , p )
2
3  switch class(A)
4      case {'int8', 'uint8'}
5          na = 8;
6      case {'int16', 'uint16'}
7          na = 16;
8      case {'int32', 'uint32'}
9          na = 32;
10     case {'int64', 'uint64'}
11         na = 64;
12     otherwise
13         error 'Multiplicands␣must␣be␣of␣the␣integer␣
               classes.'
14 end
15
16 switch class(B)
17     case {'int8', 'uint8'}
18         nb = 8;
19     case {'int16', 'uint16'}
20         nb = 16;
21     case {'int32', 'uint32'}
22         nb = 32;
23     case {'int64', 'uint64'}
24         nb = 64;
25     otherwise
26         error 'Multiplicands␣must␣be␣of␣the␣integer␣
               classes.'
27 end
28
29 n = max([na, nb]);
30
31 switch n
32     case 8
33         classname0 = 'uint8';
34         classname = 'uint16';
35         A = uint16(A(:));   B = uint16(B(:));
36     case 16
37         classname0 = 'uint16';
38         classname = 'uint32';
39         A = uint32(A(:));   B = uint32(B(:));
```

```
40      case 32
41          classname0 = 'uint32';
42          classname = 'uint64';
43          A = uint64(A(:));   B = uint64(B(:));
44      case 64
45          classname0 = 'uint64';
46          classname = 'double';
47          A = double(A(:));   B = double(B(:));
48  end
49  P = zeros(size(A),classname);
50
51  r = 1 : nb;
52  t = zeros(size(r),classname);
53  t(:) = intmax(classname0);
54  t = bitshift(t,r-1);
55  pp = t(1) * bitget(repmat(B,[1,nb]), repmat(r,[numel(B)
        ,1]));
56  pp = bitand_inexact(pp, repmat(A, [1,nb]), p, classname,
        1:n);
57  pp = bitshift(pp, repmat(r-1,[numel(B),1]));
58
59  npp = size(pp,2);
60  while npp > 2
61      j = 1;
62      for k = 1 : 3 : npp
63          if k <= (npp - 2)
64              [pp(:,j), pp(:,j+1), t(j), t(j+1)] =
                    wallace_1bit_adder_inexact_PBL( ...
65                   pp(:,k), pp(:,k+1), pp(:,k+2), t(k), t(k
                        +1), t(k+2), p);
66              npp2 = j + 1;
67              r(j:(j+1)) = r((k+1):(k+2));
68              j = j + 2;
69          elseif k == (npp - 1)
70              pp(:,j:(j+1)) = pp(:,k:(k+1));
71              npp2 = j + 1;
72              r(j:(j+1)) = r(k:(k+1));
73              t(j:(j+1)) = t(k:(k+1));
74              j = j + 2;
75          else
76              pp(:,j) = pp(:,k);
77              npp2 = j;
78              r(j) = r(k);
79              t(j) = t(k);
80              j = j + 1;
81          end
```

```
82      end
83      pp = pp(:,1:npp2);
84      r = r(1:npp2);
85      t = t(1:npp2);
86      npp = size(pp,2);
87  end
88
89  logn2 = 2 * log2(n) - 1;
90  if n <= 16
91      logn2mask1 = pow2(2*n - logn2) - 1;
92  else
93      logn2mask1 = intmax(classname);
94      for i = (2*n - logn2 + 1) : (2*n)
95          logn2mask1 = bitset(logn2mask1,i,0);
96      end
97  end
98  logn2mask2 = pow2(logn2) - 1;
99  P(:) = ling_adder_inexact_PBL(bitshift(pp(:,1),-logn2),
       bitshift(pp(:,2),-logn2), [], p);
100 P(:) = bitand(P(:),logn2mask1);
101 P(:) = bitshift(P(:),logn2);
102 P(:) = bitor(bitor(P(:),bitand(pp(:,1),logn2mask2)),bitand
       (pp(:,2),logn2mask2));
```

3.2.2 1-Bit Adder Subfunction.

```
1   function [ S, Cout, Smask, Coutmask ] =
       wallace_1bit_adder_inexact_PBL( ...
2       A, B, Cin, Amask, Bmask, Cinmask, p )
3
4   fullmask = bitand(bitand(Amask,Bmask),Cinmask);
5   halfmask1 = bitand(bitand(Amask,Bmask),bitcmp(Cinmask));
6   halfmask2 = bitand(bitand(Amask,Cinmask),bitcmp(Bmask));
7   halfmask3 = bitand(bitand(Bmask,Cinmask),bitcmp(Amask));
8   Smask = bitor(bitor(Amask,Bmask),Cinmask);
9   Coutmask = bitshift(bitor(bitor(fullmask,halfmask1),bitor(
       halfmask2,halfmask3)),1);
10  noaddmask = bitand(Smask,majority(bitcmp(Amask),bitcmp(
       Bmask),bitcmp(Cinmask)));
11  AxorB = bitand(bitxor_inexact(A,B,p),fullmask);
12  Sn = bitand(bitor_inexact(bitor_inexact(A,B,p),Cin,p),
       noaddmask);
13  Sf = bitand(bitxor_inexact(AxorB,Cin,p),fullmask);
14  S1 = bitand(bitxor_inexact(A,B,p),halfmask1);
15  S2 = bitand(bitxor_inexact(A,Cin,p),halfmask2);
16  S3 = bitand(bitxor_inexact(B,Cin,p),halfmask3);
```

```
17  Coutf = bitand(bitor_inexact(bitand_inexact(AxorB,Cin,p),
        ...
18       bitand_inexact(A,B,p),p),fullmask);
19  Cout1 = bitand(bitand_inexact(A,B,p),halfmask1);
20  Cout2 = bitand(bitand_inexact(A,Cin,p),halfmask2);
21  Cout3 = bitand(bitand_inexact(B,Cin,p),halfmask3);
22  S = bitor(bitor(bitor(Sf,S1),bitor(S2,S3)),Sn);
23  Cout = bitshift(bitor(bitor(Coutf,Cout1),bitor(Cout2,Cout3
        )),1);
```

3.3 Baugh-Wooley Multiplier

The Baugh-Wooley multiplier is capable of directly multiplying signed integers.

```
function [ P ] = multiplier_baugh_wooley_inexact_PBL( A, B
    , p )

switch class(A)
    case {'int8', 'uint8'}
        na = 8;
    case {'int16', 'uint16'}
        na = 16;
    case {'int32', 'uint32'}
        na = 32;
    case {'int64', 'uint64'}
        na = 64;
    otherwise
        error 'Multiplicands must be of the integer
           classes.'
end

switch class(B)
    case {'int8', 'uint8'}
        nb = 8;
    case {'int16', 'uint16'}
        nb = 16;
    case {'int32', 'uint32'}
        nb = 32;
    case {'int64', 'uint64'}
        nb = 64;
    otherwise
        error 'Multiplicands must be of the integer
           classes.'
end

switch class(A)
    case {'int8', 'int16', 'int32', 'int64'}
        sa = true;
    case {'uint8', 'uint16', 'uint32', 'uint64'}
        sa = false;
end

switch class(B)
    case {'int8', 'int16', 'int32', 'int64'}
        sb = true;
    case {'uint8', 'uint16', 'uint32', 'uint64'}
```

```
40          sb = false;
41  end
42
43  if (na ~= nb) && (sa ~= sb)
44      error 'If␣using␣signed␣integers,␣both␣multiplicands␣
            must␣be␣of␣the␣same␣class.'
45  end
46
47  n = na + nb;
48  if n <= 8
49      classname = 'int8';
50  elseif n <= 16
51      classname = 'int16';
52  elseif n <= 32
53      classname = 'int32';
54  elseif n <= 64
55      classname = 'int64';
56  else
57      classname = 'double';
58  end
59
60  if ~(sa || sb)
61      classname = regexprep(classname,'int','uint');
62  end
63
64  if sa
65      0x7FFF = intmax(class(A));
66      0x8000 = intmin(class(A));
67      0xFFFF = -ones('like',A);
68  else
69      0x8000 = bitset(0,na,class(A));
70      0xFFFF = intmax(class(A));
71      0x7FFF = bitxor(0xFFFF,0x8000);
72  end
73
74  if ~exist('p','var')
75      p = 1;
76  end
77
78  P = zeros(size(A),classname);
79  bb = zeros(size(A),'like',A);
80
81  bb(:) = bitget(B(:),1);
82  ss = bitand_inexact(A(:),0xFFFF*bb(:),p,class(A));
83  if sa
```

```
84      ss = bitxor(ss,0x8000);                    % NAND at
           uppermost bit
85  end
86  P(:) = bitget(ss,1);
87  ss = bitshift(ss,-1);
88  ss = bitand(ss,0x7FFF);                        % 0 at
       uppermost bit
89
90  bb(:) = bitget(B(:),2);
91  ab = bitand_inexact(A(:),0xFFFF*bb(:),p,class(A));
92  if sa
93      ab = bitxor(ab,0x8000);                    % NAND at
           uppermost bit
94  end
95  cc = bitand_inexact(ab,ss,p,class(A),1:(na-1));       %
       half adder: ab + ss
96  ss = bitxor_inexact(ab,ss,p,class(A),1:(na-1));
97  P(:) = bitset(P(:),2,bitget(ss,1));
98  ss = bitshift(ss,-1);
99  ss = bitand(ss,0x7FFF);                        % 0 at
       uppermost bit
100
101 for i = 3 : nb
102     bb(:) = bitget(B(:),i);
103     ab = bitand_inexact(A(:),0xFFFF*bb(:),p,class(A));
104     if sa
105         ab = bitxor(ab,0x8000);                % NAND at
               uppermost bit
106     end
107
108     if (i == nb) && sb
109         ab = bitxor(ab,0xFFFF);        % flip all bits (i.e
               . 1's complement)
110     end
111
112     dd = bitxor_inexact(ab,ss,p,class(A),1:(na-1));   %
           full adder: ab+ss+cc
113     ee = bitand_inexact(ab,ss,p,class(A),1:(na-1));
114     ff = bitand_inexact(dd,cc,p,class(A));
115     ss = bitxor_inexact(dd,cc,p,class(A));
116     cc = bitor_inexact(ee,ff,p,class(A));
117
118     P(:) = bitset(P(:),i,bitget(ss,1));
119     ss = bitshift(ss,-1);
120
121     if (i < nb) || (~sa)
```

```
122             ss = bitand(ss,0x7FFF);                 % 0 at
                    uppermost bit
123         else
124             ss = bitor(ss,0x8000);                  % 1 at
                    uppermost bit
125         end
126     end
127
128     dd = bitxor_inexact(cc,ss,p,class(A),2:(na-1));
129     ee = bitand_inexact(cc,ss,p,class(A),2:(na-1));
130     c = ones('like',A);
131     c(:) = sb;
132
133     for i = 1 : na
134         d = bitget(dd,i);
135         e = bitget(ee,i);
136         f = and_inexact(c,d,p);
137         s = xor_inexact(c,d,p);
138         c = or_inexact(e,f,p);
139         P(:) = bitset(P(:),nb+i,s);
140     end
```

Appendix D. Inexact Floating-Point Multiplier

```
function [ Sp, Ep, Mp ] = Multiplier_floating_inexact( Sa,
    Ea, Ma, Sb, Eb, Mb, fmt, p )

switch upper(fmt)
    case 'BINARY16'
        ne = 5;
        nm = 10;
    case 'BINARY32'
        ne = 8;
        nm = 23;
    case 'BINARY64'
        ne = 11;
        nm = 52;
    case 'BINARY128'
        ne = 15;
        nm = 112;
    otherwise
        error 'fmt must be binary16, binary32, binary64, or binary128.'
end

if ~exist('p','var')
    p = 1;
end

OxFF = cast(pow2a(ne,'uint64') - 1, 'like', Ea);
Ox7FFFFF = cast(pow2a(nm,'uint64') - 1, 'like', Ma);

% compute sign bit
Sp1 = sign_logic(Sa,Sb,p);

% compute mantissa
Ma1 = prepend_mantissa( Ea, Ma, ne, nm, p );
Mb1 = prepend_mantissa( Eb, Mb, ne, nm, p );
[Mp1, c] = multiply_mantissas (Ma1, Mb1, nm, p);

% compute exponent
[Ep1, u, v] = add_exponents ( Ea, Eb, ne, c, p );

% underflow and overflow
u_ =  cast(u,'like',Ea) * OxFF;
u__ = cast(u,'like',Ma) * Ox7FFFFF;
v_ =  cast(v,'like',Ea) * OxFF;
v__ = cast(v,'like',Ma) * Ox7FFFFF;
```

```
43
44  % output Inf in case of overflow
45  Ep2 = mux2_inexact(v_,Ep1,0xFF,p,class(Ep1),1:ne);
46  Mp2 = mux2_inexact(v__,Mp1,zeros('like',Mp1),p,class(Mp1)
        ,1:nm);
47
48  % output 0 in case of underflow
49  Sp = mux2_inexact(u,Sp1,false,p,'logical',1);
50  Ep = mux2_inexact(u_,Ep2,zeros('like',Ep2),p,class(Ep2),1:
        ne);
51  Mp = mux2_inexact(u__,Mp2,zeros('like',Mp2),p,class(Mp2)
        ,1:nm);
52
53  end
54
55  function Sp = sign_logic ( Sa, Sb, p )
56
57  Sp = xor_inexact(Sa,Sb,p);
58
59  end
60
61  function [Ep, u, v] = add_exponents ( Ea, Eb, ne, c, p )
62
63  [~, ~, E0] = adder_inexact_PBL('RC', ne, Ea, Eb, c, p);
64  v = and_inexact(bitget(E0,ne),bitget(E0,ne+1),p);      %
        overflow
65  Ep = zeros(size(E0),'like',Ea);
66  0xFFFF = bitcmp0(zeros('like',E0),ne);
67  minus127 = zeros('like',E0);
68  minus127(:) = pow2(ne) + pow2(ne-1) + 1;               %
        bin2dec('110000001')
69  E0 = adder_inexact_PBL('RC', ne+1, E0, minus127, [], p);
70  u = logical(bitget(E0,ne+1));                          %
        underflow
71  Ep(:) = bitand(E0,0xFFFF);
72
73  end
74
75  function Ma1 = prepend_mantissa( Ea, Ma, ne, nm, p )
76  %For all nonzero Ea, a 1 is prepended to the mantissa Ma.
        For Ea==0, the
77  %mantissa is returned unchanged.
78
79  H = any_high_bits_inexact_PBL(Ea,ne,p);
80
81  switch class(Ma)
```

```
82      case 'uint8'
83          if nm == 8
84              Ma = uint16(Ma);
85          end
86      case 'uint16'
87          if nm == 16
88              Ma = uint32(Ma);
89          end
90      case 'uint32'
91          if nm == 32
92              Ma = uint64(Ma);
93          end
94      case 'uint64'
95          if nm == 64
96              Ma = double(Ma);
97          end
98  end
99
100 Ma1 = bitset(Ma,nm+1,H);
101
102 end
103
104 function [ Mp, c ] = multiply_mantissas ( Ma1, Mb1, nm, p
        )
105
106 Mp0 = Multiplier_basic_inexact( Ma1, Mb1, p );
107 c = zeros(size(Mp0),'like',Mp0);
108 c(:) = bitget(Mp0,2*(nm+1));                    % carry-
        out bit
109 Mp0 = bitshifter_inexact_PBL(Mp0,-int8(c),2*(nm+1),1,p);
110
111 OxFFFF = bitcmp0(zeros('like',Mp0),nm-1);
112 lsb0 = logical(bitget(Mp0,nm+1));
113 roundbit0 = logical(bitget(Mp0,nm));
114 stickybit0 = any_high_bits_inexact_PBL(bitand(Mp0,OxFFFF),
        nm-1,p);
115 Mp0 = cast(bitshift(Mp0,-nm),'like',Ma1);
116 Mp0 = adder_inexact_PBL('RC', nm+2, Mp0, zeros('like',Mp0)
        , ...
117      and(roundbit0,or(lsb0,stickybit0)), p);
118
119 Mp = zeros(size(Mp0),'like',Ma1);
120 OxFFFF = bitcmp0(zeros('like',Mp0),nm);
121 Mp(:) = bitand(Mp0,OxFFFF);
122 c = logical(c);
123
```

```
124   end
```

Appendix E. Inexact Matrix Multiplier

```
1  function [ C, nc ] = mtimes_inexact_PBL( A, B, na, nb, p,
       bit )
2  %Performs matrix multiplication, similar to the Matlab *
3  %(asterisk) operator or mtimes function, except inexact
4  %adders and multipliers are used in the process.  The
       inputs
5  %A and B must be of the integer classes.  Inputs A and B
       can
6  %have more than two dimensions.
7  %
8  %Inputs:
9  %    A, B:  (integer arrays) Matrices to be multiplied.
10 %size(A,2) must equal size(B,1).  For higher dimensions,
11 %size(A,3) must equal size(B,3);  size(A,4) must equal
12 %size(B,4) etc.
13 %
14 %    na:  (integer) Number of bits needed to store the
       data
15 %in A.
16 %    nb:  (integer) Number of bits needed to store the
       data
17 %in B.
18 %    p:  (real number) Probability of correctness of each
19 %binary operation within the inexact adders and
       multipliers.
20 %    bit:  (integer or integer array) The highest-order
       bit
21 %which can be inexact within the addition and
       multiplication
22 %operations.  This can either be a scalar, or else an
       array
23 %with dimensions [size(A,1),size(B,2),size(A,3),size(A,4)]
24 %etc.
25 %
26 %Outputs:
27 %
28 %    C:  (integer array) Matrix product of A and B.  Dimen
       -
29 %sions of C are [size(A,1),size(B,2),size(A,3),size(A,4)]
30 %etc.  C may be of a different integer class than A and B.
31 %    nc:  (integer) Number of bits needed to store the
       data
32 %in C.  nc = na + nb + ceil(log2(size(A,2))).
33
```

```
34  if ~exist('bit','var')
35      bit = Inf;
36  end
37
38  if isscalar(na)
39      na = repmat(na,[size(A,1),size(B,2)]);
40  end
41
42  if isscalar(nb)
43      nb = repmat(nb,[size(A,1),size(B,2)]);
44  end
45
46  if isscalar(bit)
47      bit = repmat(bit,[size(A,1),size(B,2)]);
48  end
49
50  switch class(A)
51      case {'int8', 'int16', 'int32', 'int64'}
52          sa = true;
53      case {'uint8', 'uint16', 'uint32', 'uint64'}
54          sa = false;
55  end
56
57  switch class(B)
58      case {'int8', 'int16', 'int32', 'int64'}
59          sb = true;
60      case {'uint8', 'uint16', 'uint32', 'uint64'}
61          sb = false;
62  end
63
64  if (sa ~= sb) && ~strcmp(class(A),class(B))
65      error 'If␣using␣signed␣integers,␣both␣multiplicands␣
          must␣be␣of␣the␣same␣class.'
66  end
67
68  nc = na + nb + ceil(log2(size(A,2)));
69  nc(nc>64) = 64;
70  ncmax = max(nc(:));
71  if ncmax <= 8
72      classname = 'int8';
73  elseif ncmax <= 16
74      classname = 'int16';
75  elseif ncmax <= 32
76      classname = 'int32';
77  else
78      classname = 'int64';
```

```
79  end
80
81  if ~(sa || sb)
82      classname = regexprep(classname,'^int','uint');
83  end
84
85  if (~isscalar(A)) && (~isscalar(B)) && (size(A,2) ~= size(
        B,1))
86      error 'Inner␣matrix␣dimensions␣must␣agree.'
87  end
88
89  C = zeros(size(A,1),size(B,2),size(A,3),size(A,4),
        classname);
90
91  for r = 1 : size(A,1)
92      for c = 1 : size(B,2)
93          A_ = permute(A(r,:,:,:),[2,1,3,4]);
94          Cc = Multiplier_basic_inexact(A_, B(:,c,:,:), na(r
                ,c), nb(r,c), p, bit(r,c));
95          OxFFFF = cast(-1,'like',Cc);
96          OxF000 = bitshift(OxFFFF,na(r,c)+nb(r,c));
97          Ox0FFF = bitcmp(OxF000);
98          scc = (Cc < 0);
99          Cc(scc) = bitor(Cc(scc),OxF000);
100         Cc(~scc) = bitand(Cc(~scc),Ox0FFF);
101         C(r,c,:,:) = Cc(1,1,:,:);
102
103         for i = 2 : size(Cc,1)
104             nc_ = na(r,c) + nb(r,c) + ceil(log2(i));
105             C(r,c,:,:) = Adder_RCA_inexact(nc_,C(r,c,:,:),
                    cast(Cc(i,1,:,:),'like',C),[],p,1:min([nc_,
                    bit(r,c)]));
106         end
107     end
108 end
```

Appendix F. Inexact JPEG Compression Algorithm

6.1 Main Program

```
% Source:
% http://www.impulseadventure.com/photo/jpeg-huffman-coding.html

clear; close all
q = 100;

for f = 1 : 1
if f == 1
    fname = 'F16';
elseif f == 2
    fname = 'Lena';
elseif f == 3
    fname = 'Frisco';
else
    fname = 'Mandrill';
end

ncomponents = 1;                                % 3 for color, 1 for grayscale
A = imread([fname,'.tif']);
% A = A(263:342,247:326,:);      % Lena
% A = A(18:97,137:216,:);        % Mandrill
% A = A(522:681,357:516,:);      % Frisco
height = size(A,1);
width = size(A,2);
R = A(:,:,1); G = A(:,:,2); B = A(:,:,3);

fprintf('Color space transformation...\n')
Yexact = YCbCr(R,G,B);
figure; image(repmat(Yexact,[1,1,3])); title('Original Image'); axis equal; axis tight; axis off
[Y, Cb, Cr] = YCbCr_inexact(R, G, B, 1);
figure; image(repmat(Y,[1,1,3])); title('Color Space Transformation'); axis equal; axis tight; axis off
uncompressed_size = numel(Y);
npixels = numel(Y);

Y1 = tile8x8(Y);
Y1 = int8(int16(Y1) - 128);

for p = [0.99,0.999,0.9999,0.99999,0.999999]
```

```
39
40  fprintf('Discrete␣cosine␣transformation...\n')
41  Byexact = DCT0(double(Y1));
42  By = DCT_inexact_PBL(Y1,22,p);
43
44  Qy = quantize0(double(By), q, 'Y');
45
46  scandata = run_amp_huff_all(Qy,[],[],fname,q);
47
48  Aj = imread([fname,'.jpg']);
49  figure
50  if ncomponents > 1
51      image(Aj)
52  else
53      image(repmat(Aj,[1,1,3]))
54      err = double(Aj) - double(Yexact);
55  end
56  title('Final␣JPEG␣Image')
57  axis equal; axis tight; axis off
58
59  compressed_size = numel(scandata);
60  compression_ratio = uncompressed_size / compressed_size;
61  bits_per_pixel = 8 / compression_ratio;
62  err_rms = sqrt(mean(err(:).^2));
63  snr_dB = 10 * log10((double(max(Yexact(:))) - double(min(
       Yexact(:)))) ...
64      / err_rms);
65  fprintf('p␣=␣%8.6f\n',p)
66  fprintf('Uncompressed␣size:␣␣%d␣␣bytes\n',
       uncompressed_size)
67  fprintf('Compressed␣size:␣␣␣␣%d␣␣bytes\n',compressed_size)
68  fprintf('Compression␣ratio:␣␣%5.2f\n',compression_ratio)
69  fprintf('Bits␣per␣pixel:␣␣␣␣%4.2f\n',bits_per_pixel)
70  fprintf('RMS␣error:␣␣␣␣␣␣␣␣␣%7.3f\n',err_rms)
71  fprintf('SNR:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%7.3f␣␣dB\n',snr_dB)
72
73  fname2 = sprintf('%s_p=%8.6f_err=%7.3f_comp=%5.2f_snr=%7.3
       f',fname,p,err_rms,compression_ratio,snr_dB);
74  fname2 = regexprep(fname2,'\s+','');
75  fname2 = regexprep(fname2,'\.','_');
76  movefile([fname,'.jpg'],[fname2,'.jpg']);
77
78  end
79  end
```

6.2 Color Space Transformation

6.2.1 Exact Color Space Transformation.

The exact color space transformation is needed in order to compute the errors of the inexact color space transformation.

```
% Color space transformation

% Reference:
% http://www.jpeg.org/public/jfif.pdf

function [ Y, Cb, Cr ] = YCbCr( R, G, B, approx )


if ~exist('approx','var')
    approx = 'exact';
end

switch lower(approx)
    case 'uint8'
        R = uint16(R);
        G = uint16(G);
        B = uint16(B);
        Y = uint8(bitshift(154*R,-9)) + uint8(bitshift
           (151*G,-8)) + uint8(bitshift(234*B,-11));
        R = int16(R);
        G = int16(G);
        B = int16(B);
        Cb = int8(bitshift(-86*R,-9)) + int8(bitshift(-84*
           G,-8)) + int8(bitshift(B,-1));
        i = (Cb >= 0);
        Cb = uint8(bitset(Cb,8,0));
        Cb(i) = Cb(i) + uint8(128);
        Cr = int8(bitshift(R,-1)) + int8(bitshift(-107*G
           ,-8)) + int8(bitshift(-83*B,-10));
        i = (Cr >= 0);
        Cr = uint8(bitset(Cr,8,0));
        Cr(i) = Cr(i) + uint8(128);
    otherwise      % exact method
        R = double(R);
        G = double(G);
        B = double(B);
        Y = 0.299 * R  + 0.587 * G  + 0.114 * B;
        Cb = - 0.16874 * R - 0.33126 * G + 0.5 * B + 128;
```

```
36          Cr = 0.5 * R - 0.41869 * G - 0.08131 * B + 128;
37          Y = uint8(Y);
38          Cb = uint8(Cb);
39          Cr = uint8(Cr);
40  end
```

6.2.2 Inexact Color Space Transformation.

```
% Color space transformation

% Reference:
% http://www.jpeg.org/public/jfif.pdf

function [ Y, Cb, Cr ] = YCbCr_inexact( R, G, B, p )


if ~exist('p','var')
    p = 1;
end

R = uint8(R);
G = uint8(G);
B = uint8(B);
Y1 = bitshift(Multiplier_basic_inexact(R,repmat(uint8(154)
   ,size(R)),8,8,p),-9);
    % always < 77   (7 bits)
Y1 = uint8(bitand(Y1,uint16(127)));
                              % lower 7 bits
Y2 = bitshift(Multiplier_basic_inexact(G,repmat(uint8(151)
   ,size(G)),8,8,p),-8);
    % always < 151  (8 bits)
Y2 = uint8(bitand(Y2,uint16(255)));
                              % lower 8 bits
Y3 = bitshift(Multiplier_basic_inexact(B,repmat(uint8(234)
   ,size(B)),8,8,p),-11);
    % always < 30   (5 bits)
Y3 = uint8(bitand(Y3,uint16(31)));
                               % lower 5 bits
Y = Adder_RCA_inexact(8,Y1,Y2,[],p);
Y = Adder_RCA_inexact(8,Y,Y3,[],p);
Cb1 = bitshift(Multiplier_basic_inexact(R,repmat(uint8(86)
   ,size(R)),8,7,p),-9);
    % always < 43   (6 bits)
Cb1 = uint8(bitand(Cb1,uint16(63)));
                             % lower 6 bits
Cb2 = bitshift(Multiplier_basic_inexact(G,repmat(uint8(84)
   ,size(G)),8,7,p),-8);
    % always < 84   (7 bits)
Cb2 = uint8(bitand(Cb2,uint16(127)));
                            % lower 7 bits
Cb2 = Adder_RCA_inexact(7,Cb2,Cb1,[],p,1:7,true);   %
   always <= 127
```

```
34  Cb2 = bitand(Cb2,uint8(127));
                                        % lower 7 bits
35  Cb3 = bitshift(B,-1);   % always <= 127
36  Cb3 = bitor(Cb3,128);   % add 128
37  Cb = adder_subtractor_inexact_PBL('RC',8,Cb3,Cb2,true,p);
38
39  Cr1 = bitshift(Multiplier_basic_inexact(B,repmat(uint8(83)
       ,size(B)),8,7,p),-10);
40      % always < 21   (5 bits)
41  Cr1 = uint8(bitand(Cr1,uint16(31)));
                                % lower 5 bits
42  Cr2 = bitshift(Multiplier_basic_inexact(G,repmat(uint8
       (107),size(G)),8,7,p),-8);
43      % always < 107   (7 bits)
44  Cr2 = uint8(bitand(Cr2,uint16(127)));
                               % lower 7 bits
45  Cr2 = Adder_RCA_inexact(7,Cr2,Cr1,[],p,1:7,true);   %
       always <= 127
46  Cr2 = bitand(Cr2,uint8(127));
                                        % lower 7 bits
47  Cr3 = bitshift(R,-1);   % always <= 127
48  Cr3 = bitor(Cr3,128);   % add 128
49  Cr = adder_subtractor_inexact_PBL('RC',8,Cr3,Cr2,true,p);
```

6.3 Tiling Function

```
1  function [ B ] = tile8x8( A )
2
3  % See if the dimensions of A are divisible by 8.  If not,
       pad with zeros.
4  A = padarray(A, mod(-size(A),8), 'post');
5
6  % Break A up into 8x8 tiles.  Dimensions:  yminor, ymajor,
       xminor, xmajor
7  B = reshape(A, [8, size(A,1)/8, 8, size(A,2)/8]);
8
9  % Rearrange dimensions:  xminor, yminor, xmajor, ymajor
10 B = permute(B, [3, 1, 4, 2]);
```

6.4 Discrete Cosine Transformation (DCT)

6.4.1 Exact DCT.

The exact DCT is needed in order to compute the errors of the inexact DCT.

```
1  % There are four types of DCT -- this is type II (DCT-II).
2
3  % References:
4  %
5  % http://www.whydomath.org/node/wavlets/dct.html
6  %
7  % Rao, K. R. and P. Yip. Discrete Cosine Transform:
       Algorithms, Advantages,
8  % and Applications. Academic Press, San Diego, CA, 1990, p
       37.
9
10 function [ B ] = DCT0( A )
11
12 U = 1:2:15;
13 U = ([0:7].') * U;
14 U = U * pi / 16;
15 U = 0.5 * cos(U);
16 U(1,:) = 0.25 * sqrt(2);
17
18 if ndims(A) == 2
19     B = U * A * (U.');
20 else
21     B = zeros(size(A));
22     for i = 1 : size(A,4)
23         for j = 1 : size(A,3)
24             B(:,:,j,i) = U * squeeze(A(:,:,j,i)) * (U.');
25         end
26     end
27 end
```

6.4.2 Inexact DCT.

```
% There are four types of DCT -- this is type II (DCT-II).

% References:
%
% http://www.whydomath.org/node/wavlets/dct.html
%
% Rao, K. R. and P. Yip. Discrete Cosine Transform: Algorithms, Advantages,
% and Applications.  Academic Press, San Diego, CA, 1990, p 37.

function [ B ] = DCT_inexact_PBL( A, nbits, p )

% Maximum number of bits used by the DCT in all of the following images:
% Mandrill (4.2.03), Lena (4.2.04), F16 (4.2.05), and San Francisco (2.2.15):

By1max = [17 17 17 17 17 17 17 17; 16 16 16 16 16 16 16 16; ...
          16 16 16 16 16 16 16 16; 16 16 16 16 15 15 15 16;
             ...
          15 15 15 15 15 15 15 15; 15 15 15 15 15 15 15 15;
             ...
          15 15 14 15 15 15 14 14; 15 15 14 14 15 14 14 14];
     % for U * A, where nu=7 and na=8

By2max = [25 25 24 24 23 23 23 23; 25 24 24 23 23 23 23 23; ...
          24 24 23 23 23 22 23 22; 24 23 23 23 23 23 22 22;
             ...
          23 23 23 22 23 22 22 22; 23 22 22 22 22 22 22 22;
             ...
          22 22 22 22 22 21 22 21; 22 22 22 21 22 21 21 21];
     % for (U*A)*(U.'), where nu=7, na=8 and nut=7

Bymax = [11 11 10 10 9 9 9 9; 11 10 10 9 9 9 9 9; ...
         10 10 9 9 9 8 9 8; 10 9 9 9 9 9 8 8; ...
         9 9 9 8 9 8 8 8; 9 8 8 8 8 8 8 8; ...
         8 8 8 8 8 7 8 7; 8 8 8 7 8 7 7 7];
     % for the final (U*A)*(U.')

if ~exist('nbits','var')
     nbits = 22;
```

```
35  end
36
37  if ~exist('p','var')
38      p = 1;
39  end
40
41  if nbits < 18
42      na = floor((nbits + 1) / 3) + 1;
43      nu = floor(nbits / 3);
44      nut = nbits - na - nu;
45  else
46      na = 8;
47      nu = ceil((nbits - na) / 2);
48      nut = nbits - na - nu;
49  end
50
51  sa = na - 8;
52  su = nu;
53  sut = nut;
54
55  A = floor(A * pow2(sa));
56
57  U = 1:2:15;
58  U = ((0:7).') * U;
59  U = U * pi / 16;
60  U = 0.5 * cos(U);
61  U(1,:) = 0.25 * sqrt(2);
62  U1 = round(pow2(su) * U);
63  U2 = round(pow2(sut) * U.');
64
65  if nu < 8
66      U1 = int8(U1);
67  elseif nu < 16
68      U1 = int16(U1);
69  elseif nu < 32
70      U1 = int32(U1);
71  else
72      U1 = int64(U1);
73  end
74
75  if nut < 8
76      U2 = int8(U2);
77  elseif nut < 16
78      U2 = int16(U2);
79  elseif nut < 32
80      U2 = int32(U2);
```

```
81  else
82      U2 = int64(U2);
83  end
84
85  if ~ismatrix(A)
86      U1 = repmat(U1,[1,1,size(A,3),size(A,4)]);
87      U2 = repmat(U2,[1,1,size(A,3),size(A,4)]);
88  end
89
90  By1max = By1max + nu + na - 15;
91  [B,nb] = mtimes_inexact_PBL(U1,A,nu+1,na,p,By1max-3);
92  By1max = repmat(By1max,[1,1,size(B,3),size(B,4)]);
93  sgn = (B < 0);
94  B = bitset(B,By1max,sgn);
95  B = correct_upperbits(B,By1max);
96
97  B = bitshift(B,-8);
98  By2max = By2max - 8;
99
100 By2max = By2max + nut - 7;
101 B = mtimes_inexact_PBL(B,U2,nb,nut,p,By2max-5);
102 By2max = repmat(By2max,[1,1,size(B,3),size(B,4)]);
103 sgn = (B < 0);
104 B = bitset(B,By2max,sgn);
105 B = correct_upperbits(B,By2max);
106
107 B = bitshift(B,-sa-su-sut+8);
108 Bymax = repmat(Bymax,[1,1,size(B,3),size(B,4)]);
109 sgn = (B < 0);
110 B = bitset(B,Bymax,sgn);
111 B = correct_upperbits(B,Bymax);
```

6.5 Quantization

```
% References:
% http://www.ams.org/samplings/feature-column/fcarc-image-compression
% http://www.whydomath.org/node/wavlets/quantization.html

function [ Q, DQT ] = quantize0( B, q, fun )

if q < 1
    q = 1;
end

if nargin < 3
    fun = 'luminance';
end

switch upper(fun)
    case {'CHROMINANCE','CB','CR'}
        Z = [17 18 24 47 99 99 99 99; 18 21 26 66 99 99 99 99; ...
            24 26 56 99 99 99 99 99; 47 66 99 99 99 99 99 99; ...
            99 99 99 99 99 99 99 99; 99 99 99 99 99 99 99 99; ...
            99 99 99 99 99 99 99 99; 99 99 99 99 99 99 99 99].';
    otherwise
        Z = [16 11 10 16 24 40 51 61; 12 12 14 19 26 58 60 55; ...
            14 13 16 24 40 57 69 56; 14 17 22 29 51 87 80 62; ...
            18 22 37 56 68 109 103 77; 24 35 55 64 81 104 113 92; ...
            49 64 78 87 103 121 120 101; 72 92 95 98 112 100 103 99].';
end

if q <= 50
    alpha = 50 / q;
else
    alpha = 2 - q / 50;
end

if q >= 100
    Q = round(B);
```

```
36      DQT = ones(size(Z));
37  else
38      DQT = round(alpha * Z);
39      if ~ismatrix(B)
40          Z = repmat(Z,[1, 1, size(B,3), size(B,4)]);
41      end
42      Q = round(B ./ (alpha * Z));
43  end
44
45  if all((abs(Q(:)) <= 2047) & (~isnan(Q(:))))
46      Q = int16(Q);
47  else
48      warning('Quantized data out of range.')
49      Q((abs(Q)>2047) | isnan(Q)) = 0;
50      Q = int16(Q);
51  end
```

6.6 Zigzag Function

```
% Reference:
% http://www.ams.org/samplings/feature-column/fcarc-image-compression

function [ Q1 ] = zigzag8x8( Q )

zigzag = [ 1,  2,  6,  7, 15, 16, 28, 29 ;
           3,  5,  8, 14, 17, 27, 30, 43 ;
           4,  9, 13, 18, 26, 31, 42, 44 ;
          10, 12, 19, 25, 32, 41, 45, 54 ;
          11, 20, 24, 33, 40, 46, 53, 55 ;
          21, 23, 34, 39, 47, 52, 56, 61 ;
          22, 35, 38, 48, 51, 57, 60, 62 ;
          36, 37, 49, 50, 58, 59, 63, 64 ].';

Q1 = zeros(size(Q), class(Q));

if ismatrix(Q)              % two-dimensional, one 8x8 tile only
    Q1(zigzag) = Q;
else                        % four-dimensional, entire image
    [~,z2,z3] = ndgrid(1:64, (0:(size(Q,3)-1))*64, (0:(size(Q,4)-1))*64*size(Q,3));
    z2 = reshape(z2, size(Q));
    z3 = reshape(z3, size(Q));
    zigzag = repmat(zigzag, [1, 1, size(Q,3),size(Q,4)]) + z2 + z3;
    Q1(zigzag) = Q;
end
```

6.7 Run-Amplitude Encoding

```
% Run-amplitude encoding

% References:
% http://cnx.org/content/m11096/latest/
% http://www.impulseadventure.com/photo/jpeg-huffman-coding.html

function [ jpg, jpgstr ] = run_amp_encode0( Q1, bitstrings )

% bitstrings must be sorted in order by code number.

if numel(Q1) > 1
    dc = false;
else
    dc = true;
end
jpg = cell(2,numel(Q1));

nz = 0;
j = 0;
for i = 1 : numel(Q1)
    if Q1(i) || dc
        j = j + 1;
        s = numel(dec2bin(abs(Q1(i))));
        if (16 * nz + s + 1) <= numel(bitstrings)
            jpg{1,j} = bitstrings{16 * nz + s + 1};
        else
            jpg{1,j} = bitstrings{1};
            warning 'Data out of range.'
        end
        if Q1(i) >= 0
            jpg{2,j} = dec2bin(Q1(i));
        else
            jpg{2,j} = dec2bin(bitcmp0(-Q1(i),s),s);
        end
        nz = 0;
    else
        nz = nz + 1;
        if (i == numel(Q1)) || (~sum(abs(Q1(i:end))))
            j = j + 1;
            jpg{1,j} = bitstrings{1};      % end-of-block (EOB)
            if j < numel(Q1)
```

```
42                      jpg(:,(j+1):end) = [];
43                  end
44                  break
45              elseif nz == 16
46                  j = j + 1;
47                  jpg{1,j} = bitstrings{241};  % zero run length
                        (ZRL) (0xF0 + 1)
48                  nz = 0;
49              end
50          end
51      end
52
53      if nargout >= 2
54          jpgstr = char('1' * ones([1,16*numel(jpg)],'uint8'));
55          k = 1;
56          for m = 1 : numel(jpg)
57              jpgstr(k:(k+numel(jpg{m})-1)) = jpg{m};
58              k = k + numel(jpg{m});
59          end
60          jpgstr = jpgstr(1:(k-1));
61      end
```

6.8 Huffman Encoding

```
function scandata = run_amp_huff_all( Qy, Qcb, Qcr, fname, q )
%Source:
%http://www.impulseadventure.com/photo/jpeg-huffman-coding.html

if ~exist('Qy','var')
    Qy = [];
end

if ~exist('Qcb','var')
    Qcb = [];
end

if ~exist('Qcr','var')
    Qcr = [];
end

[~, fname] = fileparts(fname);
fname = [fname,'.jpg'];

ncomponents = (~isempty(Qy)) + (~isempty(Qcb)) + (~isempty(Qcr));
height = 8 * size(Qy,4);
width = 8 * size(Qy,3);
Q0 = zigzag8x8(Qy);

load huffman_dc_luminance_sorted
bitstrings_dc_luminance = bitstrings;
load huffman_ac_luminance_sorted
bitstrings_ac_luminance = bitstrings;
load huffman_dc_chrominance_sorted
bitstrings_dc_chrominance = bitstrings;
load huffman_ac_chrominance_sorted
bitstrings_ac_chrominance = bitstrings;

fprintf('Run-amplitude encoding...\n')
jpgstr = char('1' * ones([1,32*numel(Qy)],'uint8'));
k = 1;
prev_Qy_dc = 0;
prev_Qcb_dc = 0;
prev_Qcr_dc = 0;
dc_correction = 0;
for i = 1 : size(Qy,4)
```

```
42      fprintf('%i␣', size(Qy,4)-i);
43      if ~mod(i,10)
44          fprintf('\n');
45      end
46      for j = 1 : size(Qy,3)
47          Q1 = squeeze(Q0(:,:,j,i));
48          [~, jpgstr0] = run_amp_encode0(Q1(1)-prev_Qy_dc,
               bitstrings_dc_luminance);
49          jpgstr(k:(k+numel(jpgstr0)-1)) = jpgstr0;
50          k = k + numel(jpgstr0);
51          [~, jpgstr0] = run_amp_encode0(Q1(2:end),
               bitstrings_ac_luminance);
52          jpgstr(k:(k+numel(jpgstr0)-1)) = jpgstr0;
53          k = k + numel(jpgstr0);
54          prev_Qy_dc = Q1(1);
55
56          % Very slight error in the dc component -- not
               sure why.
57          % No big deal for small images, but for large
               images it accumulates
58          prev_Qy_dc = prev_Qy_dc - floor(dc_correction);
59          if floor(dc_correction) >= 1
60              dc_correction = 0;
61          else
62              dc_correction = dc_correction + 0.02;
63          end
64
65          if ncomponents > 1
66              Q1 = zigzag8x8(squeeze(Qcb(:,:,j,i)));
67              [~, jpgstr0] = run_amp_encode0(Q1(1)-
                   prev_Qcb_dc, bitstrings_dc_chrominance);
68              jpgstr(k:(k+numel(jpgstr0)-1)) = jpgstr0;
69              k = k + numel(jpgstr0);
70              [~, jpgstr0] = run_amp_encode0(Q1(2:end),
                   bitstrings_ac_chrominance);
71              jpgstr(k:(k+numel(jpgstr0)-1)) = jpgstr0;
72              k = k + numel(jpgstr0);
73              prev_Qcb_dc = Q1(1);
74
75              Q1 = zigzag8x8(squeeze(Qcr(:,:,j,i)));
76              [~, jpgstr0] = run_amp_encode0(Q1(1)-
                   prev_Qcr_dc, bitstrings_dc_chrominance);
77              jpgstr(k:(k+numel(jpgstr0)-1)) = jpgstr0;
78              k = k + numel(jpgstr0);
79              [~, jpgstr0] = run_amp_encode0(Q1(2:end),
                   bitstrings_ac_chrominance);
```

```
80              jpgstr(k:(k+numel(jpgstr0)-1)) = jpgstr0;
81              k = k + numel(jpgstr0);
82              prev_Qcr_dc = Q1(1);
83          end
84      end
85  end
86
87  jpgstr = jpgstr(1:(k-1));
88  jpgstr = stuffbyte(jpgstr)
89  jpgstr = reshape(jpgstr, [8, numel(jpgstr)/8]).';
90  scandata = bin2dec(jpgstr);
91
92  fileID = fopen(fname,'w');
93  fwrite1_SOI( fileID );
94  fwrite2_APP0( fileID );
95  [~, DQT] = quantize0( zeros(8,8), q, 'luminance' );
96  fwrite3_DQT( fileID, 0, DQT, q, 'luminance' );
97  if ncomponents > 1
98      [~, DQT] = quantize0( zeros(8,8), q, 'chrominance' );
99      fwrite3_DQT( fileID, 1, DQT, q, 'chrominance' );
100 end
101 fwrite4_SOF0( fileID, height, width, ncomponents );
102 load('huffman_dc_luminance','DHT')
103 fwrite5_DHT( fileID, 0, 0, DHT );
104 load('huffman_ac_luminance','DHT')
105 fwrite5_DHT( fileID, 0, 1, DHT );
106 if ncomponents > 1
107     load('huffman_dc_chrominance','DHT')
108     fwrite5_DHT( fileID, 1, 0, DHT );
109     load('huffman_ac_chrominance','DHT')
110     fwrite5_DHT( fileID, 1, 1, DHT );
111 end
112 fwrite6_SOS( fileID, ncomponents );
113 fwrite(fileID,scandata);
114 fwrite(fileID, [255; 217]);      % 0xFFD9   (EOI)
115 fclose('all');
```

6.9 Stuff Byte

```
% Reference:
% http://www.impulseadventure.com/photo/jpeg-huffman-coding.html

function [ jpgstr_stuffed ] = stuffbyte( jpgstr )

% Check to see if the number of bits is a multiple of 8.
% If not, then pad with ones.
jpgstr1 = char(padarray(uint8(jpgstr), [0,mod(-numel(jpgstr),8)], ...
    uint8('1'), 'post'));

% Split jpgstr1 up into 8-bit groups.
jpgstr2 = reshape(jpgstr1,[8,numel(jpgstr1)/8]).';

jpgstr_stuffed = [];
for i = 1 : size(jpgstr2,1)
    jpgstr_stuffed = [jpgstr_stuffed, jpgstr2(i,:)];
    if strcmp(jpgstr2(i,:),'11111111')   % 0xFF becomes 0xFF00 (stuff byte)
        jpgstr_stuffed = [jpgstr_stuffed, '00000000'];
    end
end
```

6.10 File Operations

```
function [ count ] = fwrite1_SOI( fileID )

% Reference:
% http://vip.sugovica.hu/Sardi/kepnezo/JPEG%20File%20Layout%20and%20Format.htm

count = fwrite(fileID, [255; 216]);
                            % 0xFFD8    (SOI)
```

```
function [ count ] = fwrite2_APP0( fileID )

% Reference:
% http://vip.sugovica.hu/Sardi/kepnezo/JPEG%20File%20Layout%20and%20Format.htm

A = [255; 224; ...              % 0xFFE0    (APP0)
     0; 16; ...                 % APP0 field length
     74; 70; 73; 70; 0; ...     % JFIF identifier
     1; 2; ...                  % version 1.02
     1; ...                     % units=1 means dots per inch (DPI)
     0; 72; ...                 % X density (DPI)
     0; 72; ...                 % Y density (DPI)
     0; ...                     % X thumnail width
     0];                        % Y thumnail height

count = fwrite(fileID, A);
```

```
function [ count, DQT ] = fwrite3_DQT( fileID, tableID, DQT, q, fun )

% References:
% http://www.ams.org/samplings/feature-column/fcarc-image-compression
% http://www.whydomath.org/node/wavlets/quantization.html
% http://vip.sugovica.hu/Sardi/kepnezo/JPEG%20File%20Layout%20and%20Format.htm

if nargin < 5
    fun = 'luminance';
end

if nargin < 4
    q = 100;
end
```

```
15
16  if numel(DQT) ~= 64
17
18      if (q < 1) || (q > 100)
19          error 'q must be between 1 and 100.'
20      end
21
22      switch upper(fun)
23          case {'C','CB','CR','CHROMINANCE'}
24              Z = [17 18 24 47 99 99 99 99; 18 21 26 66 99
                    99 99 99; ...
25                   24 26 56 99 99 99 99 99; 47 66 99 99 99 99
                         99 99; ...
26                   99 99 99 99 99 99 99 99; 99 99 99 99 99 99
                         99 99; ...
27                   99 99 99 99 99 99 99 99; 99 99 99 99 99 99
                         99 99];
28          otherwise           % luminance
29              Z = [16 11 10 16 24 40 51 61; 12 12 14 19 26
                    58 60 55; ...
30                   14 13 16 24 40 57 69 56; 14 17 22 29 51 87
                         80 62; ...
31                   18 22 37 56 68 109 103 77; 24 35 55 64 81
                        104 113 92; ...
32                   49 64 78 87 103 121 120 101; 72 92 95 98
                        112 100 103 99];
33      end
34
35      if q <= 50
36          alpha = 50 / q;
37      else
38          alpha = 2 - q / 50;
39      end
40
41      DQT = round(alpha * Z);
42
43  end
44
45  A = [255; 219; ...                  % 0xFFDB   (DQT)
46      0; 67; ...                      % DQT field length
47      tableID];                       % destination ID number
48
49  A = [A; reshape(zigzag8x8(DQT),[64,1])];
50
51  count = fwrite(fileID, A);
```

```
function [ count ] = fwrite4_SOF0( fileID,  height, width,
    ncomponents )

% Reference:
% http://vip.sugovica.hu/Sardi/kepnezo/JPEG%20File%20
   Layout%20and%20Format.htm

A = [255; 192; ...              % 0xFFC0   (SOF0)
    0; 3*ncomponents+8; ...     % SOF0 field length
    8; ...                      % data precision
    floor(height/256); mod(height,256); ...     % image
       height
    floor(width/256); mod(width,256); ...       % image
       width
    ncomponents];               % number of components

for i = 1 : ncomponents
    A = [A; i; ...              % component ID number
        17; ...                 % 0x11   sampling factors
        i~=1];                  % quantization table number
            (0 or 1)
end

count = fwrite(fileID, A);
```

```
function [ count ] = fwrite5_DHT( fileID, tableID, acdc,
   DHT )

% Reference:
% http://vip.sugovica.hu/Sardi/kepnezo/JPEG%20File%20
   Layout%20and%20Format.htm

% DHT must be sorted in order by code length.
% DHT{i} = vector of all codes of length i bits.

if (tableID~=0) && (tableID~=1) && (tableID~=2) && (
   tableID~=3)
    error 'tableID must be 0, 1, 2, or 3.'
end

switch upper(acdc)
    case 'DC'
        acdc = 0;
    case 'AC'
        acdc = 1;
    case {0,1}
    otherwise
```

```
20          error 'acdc must be 0 (dc) or 1 (ac).'
21  end
22
23  A = [255; 196; ...                  % 0xFFC4   (DHT)
24      0; 0; ...                       % DHT field length (
            calculated below)
25      16*acdc+tableID; ...            % DHT information byte
26      zeros(16,1)];
27
28  for i = 1 : numel(DHT)
29      A(i+5) = numel(DHT{i});
30  end
31
32  for i = 1 : numel(DHT)
33      A = [A; DHT{i}(:)];
34  end
35
36  A(4) = numel(A) - 2;                % DHT field length
37
38  count = fwrite(fileID, A);
```

```
1   function [ count ] = fwrite6_SOS( fileID, ncomponents )
2
3   % Reference:
4   % http://vip.sugovica.hu/Sardi/kepnezo/JPEG%20File%20
        Layout%20and%20Format.htm
5
6   if (ncomponents~=1)&& (ncomponents~=2)&& (ncomponents~=3)
        &&(ncomponents~=4)
7       error 'ncomponents must be 1, 2, 3, or 4.'
8   end
9
10  A = [255; 218; ...                  % 0xFFDA   (SOS)
11      0; 2*ncomponents+6; ...         % SOS field length
12      ncomponents];                   % number of components
13
14  for i = 1 : ncomponents
15      A = [A; i; ...                  % component ID number
16          17*(i~=1)];                 % DC & AC Huffman table ID
                numbers
17  end
18
19  A = [A; 0; 63; ...                  % spectral selection
20      0];                             % successive approximation
21
22  count = fwrite(fileID, A);
```

Appendix G. Logical Functions

7.1 Inexact NOT

```
1  function [ B ] = not_inexact( A, p )
2  %Calculates the logical NOT of the input argument A,
       similar to the
3  %standard not function, except that each bit has a random
       error
4  %probability equal to 1-p.
5  %
6  %Inputs:
7  %A:  (logical array) Input argument for the NOT operator.
8  %p:  (scalar) Probability of correctness of each bit
       within the output B.
9  %    0 <= p <= 1.
10 %
11 %Outputs:
12 %B:  (logical array)  B = A OR B, subject to a bitwise
       random error
13 %    probability 1-p.  B has the same dimensions as A and
       B.
14 %
15 %Notes:
16 %If p=1, then B = ~A and is error-free.
17 %If p=0, then B = A.
18 %If p=0.5, then B contains completely random data.
19 %
20 %Reference:
21 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
22 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
23 %Computer Science, Jun 2008.
24
25 B = ~A;
26 err = (rand(size(B)) > p);
27 B(err) = ~B(err);
```

7.2 Inexact AND

```
1  function [ C ] = and_inexact( A, B, p )
2  %Calculates the bitwise AND of the input arguments A and B
       , similar to the
3  %standard and function, except that each bit has a random
       error
4  %probability equal to 1-p.
5  %
6  %Inputs:
7  %A, B:  (logical arrays) Input arguments for the AND
       operator.
8  %    B must have the same dimensions as A.
9  %p:  (scalar) Probability of correctness of each bit
       within the output C.
10 %    0 <= p <= 1.
11 %
12 %Outputs:
13 %C:  (logical array)  C = A AND B, subject to a bitwise
       random error
14 %    probability 1-p.  C has the same dimensions as A and
       B.
15 %
16 %Notes:
17 %If p=1, then C = A AND B and is error-free.
18 %If p=0, then C = A NAND B.
19 %If p=0.5, then C contains completely random data.
20 %
21 %Reference:
22 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
23 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
24 %Computer Science, Jun 2008.
25
26 C = and(A, B);
27 err = (rand(size(C)) > p);
28 C(err) = ~C(err);
```

7.3 Inexact OR

```
1  function [ C ] = or_inexact( A, B, p )
2  %Calculates the bitwise OR of the input arguments A and B,
       similar to the
3  %standard or function, except that each bit has a random
       error
4  %probability equal to 1-p.
5  %
6  %Inputs:
7  %A, B:  (logical arrays) Input arguments for the OR
       operator.
8  %    B must have the same dimensions as A.
9  %p:  (scalar) Probability of correctness of each bit
       within the output C.
10 %    0 <= p <= 1.
11 %
12 %Outputs:
13 %C:  (logical array)  C = A OR B, subject to a bitwise
       random error
14 %    probability 1-p.  C has the same dimensions as A and
       B.
15 %
16 %Notes:
17 %If p=1, then C = A OR B and is error-free.
18 %If p=0, then C = A NOR B.
19 %If p=0.5, then C contains completely random data.
20 %
21 %Reference:
22 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
23 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
24 %Computer Science, Jun 2008.
25
26 C = or(A, B);
27 err = (rand(size(C)) > p);
28 C(err) = ~C(err);
```

7.4 Inexact XOR

```
1  function [ C ] = xor_inexact( A, B, p )
2  %Calculates the bitwise XOR of the input arguments A and B
       , similar to the
3  %standard xor function, except that each bit has a random
       error
4  %probability equal to 1-p.
5  %
6  %Inputs:
7  %A, B:  (logical arrays) Input arguments for the XOR
       operator.
8  %    B must have the same dimensions as A.
9  %p:  (scalar) Probability of correctness of each bit
       within the output C.
10 %    0 <= p <= 1.
11 %
12 %Outputs:
13 %C:  (logical array)  C = A XOR B, subject to a bitwise
       random error
14 %    probability 1-p.  C has the same dimensions as A and
       B.
15 %
16 %Notes:
17 %If p=1, then C = A XOR B and is error-free.
18 %If p=0, then C = A XNOR B.
19 %If p=0.5, then C contains completely random data.
20 %
21 %Reference:
22 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
23 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
24 %Computer Science, Jun 2008.
25
26 C = xor(A, B);
27 err = (rand(size(C)) > p);
28 C(err) = ~C(err);
```

7.5 Inexact Multiplexer

```
1  function [ Z, Z_ ] = mux2_inexact( S, A0, A1, p, classname
       , bit )
2  %mux2_inexact:  Two-input multiplexer.  Computes the
       bitwise
3  %((~S AND A0) OR (S AND A1)), similar to the bitand func-
4  %tion, except that each AND, OR, and NOT gate has a random
5  %error probability equal to 1-p.
6  %
7  %Inputs:
8  %S:  (nonnegative integer array) Selector.
9  %A0, A1:  (nonnegative integer arrays) Input signals for
       the
10 %    multiplexer.  S, A0, and A1 must all have the same di
       -
11 %    mensions.
12 %p:  (scalar) Probability of correctness of each bit
       within
13 %    the output Z.  0 <= p <= 1.
14 %classname:  (string) The class name of the output arrays
       Z
15 %    and Z_.
16 %bit:  (integer vector) Which bit positions can be inexact
       .
17 %    Position 1 is the lowest-order bit.  (optional) If
       bit
18 %    is omitted, then all positions can be inexact.
19 %
20 %Outputs:
21 %Z:  (integer array)  Z = ((~S AND A0) OR (S AND A1)), sub
       -
22 %    ject to a bitwise random error probability 1-p.  Z
       has
23 %    the same dimensions as S, A0, and A1.
24 %Z_:  (integer array) Z_ = ((S AND A0) OR (~S AND A1)),
       sub-
25 %    ject to a bitwise random error probability 1-p.  Z_
       has
26 %    the same dimensions as S, A0, and A1.
27 %
28 %Notes:
29 %If p=1, then Z = ((~S AND A0) OR (S AND A1)) and is
30 %    error-free.
31 %If p=0, then Z = ((~~S NAND A0) NOR (S NAND A1)).
32 %If p=0.5, then Z contains completely random data.
```

```
33  %
34  %Reference:
35  %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
36  %Boolean logic and its meaning," Tech. Rep. TR-08-05, Rice
37  %University, Department of Computer Science, Jun 2008.
38
39  if ~exist('p','var')
40      p = 1;
41  end
42
43  if ~exist('classname','var')
44      classname = class(bitor(bitand(bitcmp(S(1)),A0(1)),
            bitand(S(1),A1(1))));
45  end
46
47  switch classname
48      case 'logical'
49          n = 1;
50      case {'uint8', 'int8'}
51          n = 8;
52      case {'uint16', 'int16'}
53          n = 16;
54      case {'uint32', 'int32'}
55          n = 32;
56      case {'uint64', 'int64'}
57          n = 64;
58      otherwise
59          error 'classname must be logical, uint8, uint16,
                uint32, uint64, int8, int16, int32, or int64.'
60  end
61
62  if ~exist('bit','var')
63      bit = 1 : n;
64  end
65
66  if islogical(S)
67      S_ = not_inexact(S, p);
68
69      Z0 = and_inexact(S_, A0, p);
70      Z1 = and_inexact(S, A1, p);
71      Z = or_inexact(Z0, Z1, p);
72
73      Z2 = and_inexact(S, A0, p);
74      Z3 = and_inexact(S_, A1, p);
75      Z_ = or_inexact(Z2, Z3, p);
76  else
```

```
77      S_ = bitcmp_inexact(S, n, p, classname, bit);
78
79      Z0 = bitand_inexact(S_, A0, p, classname, bit);
80      Z1 = bitand_inexact(S, A1, p, classname, bit);
81      Z = bitor_inexact(Z0, Z1, p, classname, bit);
82
83      Z2 = bitand_inexact(S, A0, p, classname, bit);
84      Z3 = bitand_inexact(S_, A1, p, classname, bit);
85      Z_ = bitor_inexact(Z2, Z3, p, classname, bit);
86  end
```

7.6 Inexact AND-OR-2-1

```
1  function [ D ] = AO21_inexact( A, B, C, p, classname, bit
       )
2  %Calculates the bitwise (A OR (B AND C)), similar to the
       bitand function,
3  %except that each bit has a random error probability equal
       to 1-p.
4  %
5  %Inputs:
6  %A, B, C:  (nonnegative integer arrays) Input arguments
       for the AOI
7  %    function.  A, B, and C must all have the same
       dimensions.
8  %p:  (scalar) Probability of correctness of each bit
       within the output D.
9  %    0 <= p <= 1.
10 %classname:  (string) The class name of the output array D
       .
11 %bit:  (integer vector) Which bit positions can be inexact
       .  Position 1 is
12 %  lowest-order bit.  (optional) If bit is omitted, then
       all positions can
13 %  be inexact.
14 %
15 %Outputs:
16 %D:  (nonnegative integer array)  D = (A OR (B AND C)),
       subject to a
17 %    bitwise random error probability 1-p.  D has the same
       dimensions as A,
18 %    B, and C.
19 %
20 %Notes:
21 %If p=1, then D = (A OR (B AND C)) and is error-free.
22 %If p=0, then D = (A NOR (B AND C))  (and-or-invert).
23 %If p=0.5, then D contains completely random data.
24 %
25 %Reference:
26 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
27 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
28 %Computer Science, Jun 2008.
29
30 D0 = bitand(B, C);
31 if nargin < 5
```

```
32      D = bitor(A, D0);
33      classname = class(D);
34  else
35      D = zeros(size(A), classname);
36      D(:) = bitor(A, D0);
37  end
38
39  if nargin >= 6
40      err = biterrors(size(D), p, classname, bit);
41  else
42      err = biterrors(size(D), p, classname);
43  end
44
45  D = bitxor(D, err);
```

7.7 Inexact AND-OR-AND-OR-2-1-1-1

```
function [ F ] = AOAO2111_inexact( A, B, C, D, E, p,
    classname, bit )
%AOAO2111_inexact:  Computes the bitwise (A OR (B AND (C
    OR (D AND E)))),
%similar to the bitand function, except that each bit has
    a random error
%probability equal to 1-p.

F0 = bitand(D, E);
F1 = bitor(C, F0);
F2 = bitand(B, F1);
if nargin < 7
    F = bitor(A, F2);
    classname = class(F);
else
    F = zeros(size(A), classname);
    F(:) = bitor(A, F2);
end

if nargin >= 8
    err = biterrors(size(F), p, classname, bit);
else
    err = biterrors(size(F), p, classname);
end

F = bitxor(F, err);
```

7.8 Inexact n-Input AND

```
1  function [ C ] = and3_inexact( A, p )
2  %Performs the logical AND of all elements along the rows
       of
3  %a two-dimensional array A.  The output is a column vector
       .
4  %This function simulates an inexact N-input AND gate (
       where
5  %N is the number of columns of A), by ANDing the inputs
6  %pairwise in a binary tree of 2-input inexact AND gates.
7  %Each 2-input AND gate has a probability of correctness p
8  %and a probability of error 1-p.
9  %
10 %Note that since it is a binary tree structure, if there
       is
11 %an odd number of inputs (that is, if N is not a power of
       2)
12 %then the rightmost columns of A are evaluated last.
       There-
13 %fore, the rightmost columns suffer from less inexactness
14 %than the rest of the array.
15 %
16 %Inputs:
17 %A:  (2-dimensional logical array) Input data.
18 %p:  (scalar) Probability of correctness of each 2-input
       AND
19 %    gate within the binary tree.  (0 <= p <= 1)
20 %
21 %Output:
22 %C:  (column vector of logicals) Approximate N-input AND
       of
23 %    each row of A.
24
25 if ~exist('p','var')
26     p = 1;
27 end
28
29 for i = 1 : ceil(log2(size(A,2)))
30     for j = 1 : size(A,2)
31         if j < size(A,2)
32             A = [A(:,1:(j-1)), and_inexact(A(:,j),A(:,j+1)
                   ,p), A(:,(j+2):end)];
33         else
34             break
35         end
```

```
36        end
37    end
38
39    C = A;
```

7.9 Inexact n-Input OR

```
1  function [ C ] = or3_inexact( A, p )
2  %Performs the logical OR of all elements along the rows of
3  %a two-dimensional array A.  The output is a column vector
       .
4  %This function simulates an inexact N-input OR gate (where
5  %N is the number of columns of A), by ORing the inputs
6  %pairwise in a binary tree of 2-input inexact OR gates.
7  %Each 2-input OR gate has a probability of correctness p
8  %and a probability of error 1-p.
9  %
10 %Note that since it is a binary tree structure, if there
       is
11 %an odd number of inputs (that is, if N is not a power of
       2)
12 %then the rightmost columns of A are evaluated last.
       There-
13 %fore, the rightmost columns suffer from less inexactness
14 %than the rest of the array.
15 %
16 %Inputs:
17 %A:  (2-dimensional logical array) Input data.
18 %p:  (scalar) Probability of correctness of each 2-input
       OR
19 %    gate within the binary tree.  (0 <= p <= 1)
20 %
21 %Output:
22 %C:  (column vector of logicals) Approximate N-input OR of
23 %    each row of A.
24
25 if ~exist('p','var')
26     p = 1;
27 end
28
29 for i = 1 : ceil(log2(size(A,2)))
30     for j = 1 : size(A,2)
31         if j < size(A,2)
32             A = [A(:,1:(j-1)), or_inexact(A(:,j),A(:,j+1),
                   p), A(:,(j+2):end)];
33         else
34             break
35         end
36     end
37 end
38
```

```
39    C = A;
```

Appendix H. Bitwise Functions

8.1 N-Bit One's Complement (Exact)

```
function [ cmp ] = bitcmp0( A, N )
%Returns an N-bit complement of the input A.
%Same as bitcmp(A,N) which is deprecated.

switch class(A)
    case {'int8','uint8'}
        nmax = 8;
    case {'int16','uint16'}
        nmax = 16;
    case {'int32','uint32'}
        nmax = 32;
    case {'int64','uint64'}
        nmax = 64;
    case {'double','single'}
        if any(A < 0) || any(A ~= floor(A))
            c = class(A);
            c(1) = upper(c(1));
            error([c,' inputs must be nonnegative integers.'])
        elseif any(A > intmax('uint64'))
            error 'Values in A should not have "on" bits in positions greater than N.'
        end
        A = uint64(A);
        nmax = 64;
    case 'logical'
        A = uint8(A);
        nmax = 1;
    otherwise
        error 'Operands to bitcmp0 must be numeric.'
end

if ~exist('N','var')
    N = nmax;
end

if any(N < 0) || any(N > nmax) || any(N ~= floor(N))
    error 'Number of bits must be an integer within the range of the input A.'
end

if N < 64
```

```
40      OxFFFF = cast(pow2a(N,'uint64') - 1, 'like', A);
41  elseif intmin(class(A)) < 0
42      OxFFFF = intmin('int64');
43  else
44      OxFFFF = intmax('uint64');
45  end
46
47  cmp = bitxor(A, OxFFFF);
```

8.2 Majority Function (Exact)

```
function [ D ] = majority( A, B, C )
%majority:  Computes the bitwise majority of A, B, and C,
%similar to the bitand function.

D0 = bitand(bitand(bitcmp(A),B),C);
D1 = bitand(bitand(A,bitcmp(B)),C);
D2 = bitand(bitand(A,B),bitcmp(C));
D3 = bitand(bitand(A,B),C);
D = bitor(bitor(bitor(D0,D1),D2),D3);

end
```

8.3 Bitwise Error Generator

This function is used by many higher functions to simulate unreliable computation.

```
1  function [ err ] = biterrors( outputsize, p, classname,
       bit )
2  %Generates an array of random integers.  The numbers are
       generated bitwise
3  %such that p is the probability that each bit is 0, and 1-
       p is the
4  %probability that each bit 1.
5  %
6  %Inputs:
7  %outputsize:  (vector) The dimensions of the output array
       err.
8  %p:  (scalar) The probability that each output bit is a
       zero.  0 <= p <= 1
9  %classname:  (string) The class name of the output array
       err.
10 %bit:  (integer vector) Which bit positions can be nonzero
       .  Position 1 is
11 %  lowest-order bit.  (optional) If bit is omitted, then
       all positions can
12 %  be nonzero.
13 %
14 %Ouptut:
15 %err:  (integer array) Random array of dimensions
       specified by outputsize.
16
17 switch classname
18     case {'uint8', 'int8'}
19         n = 8;
20      case {'uint16', 'int16'}
21         n = 16;
22     case {'uint32', 'int32'}
23         n = 32;
24     case {'uint64', 'int64'}
25         n = 64;
26     otherwise
27         error 'classname must be uint8, uint16, uint32,
              uint64, int8, int16, int32, or int64.'
28 end
29
30 err = zeros(outputsize, classname);
31 err0 = ( rand([numel(err), n]) > p );       % generate
       random binary digits
```

```
32  twos = pow2(n-1:-1:0);
33  err(:) = sum(err0 .* twos(ones(numel(err),1),:),2);  %
       convert binary to dec
34
35  if nargin >= 4
36      bit = bit((bit >= 1) & (bit <= n));
37      b = zeros(classname);
38      b(:) = sum(bitset(zeros(classname), bit(:)));
39      err = bitand(err, b);
40  end
```

8.4 *N*-Bit One's Complement (Inexact)

```
1  function [ C ] = bitcmp_inexact( A, n, p, classname, bit )
2  %Calculates the n-bit complement of the input argument A,
       similar to the
3  %standard bitcmp function, except that each bit has a
       random error
4  %probability equal to 1-p.
5  %
6  %Inputs:
7  %A:  (nonnegative integer array) Input argument for the
       bitcmp function.
8  %n:  (integer) Number of bits to complement.  The lowest n
        bits are comple-
9  %    mented.  (optional) If n is omitted, then all bits
       are complemented.
10 %p:  (scalar) Probability of correctness of each bit
       within the output C.
11 %    0 <= p <= 1.
12 %classname:  (string) The class name of the output array C
       .
13 %bit:  (integer vector) Which bit positions can be inexact
       .  Position 1 is
14 %  lowest-order bit.  (optional) If bit is omitted, then
       all positions can
15 %  be inexact.
16 %
17 %Outputs:
18 %C:  (nonnegative integer array) The bitwise complement of
        A, subject to a
19 %    bitwise random error probability 1-p.  C has the same
        dimensions as A.
20 %
21 %Notes:
22 %If p=1, then C = bitcmp(A) and is error-free.
23 %If p=0, then C = A.
24 %If p=0.5, then C contains completely random data.
25 %
26 %Reference:
27 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
28 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
29 %Computer Science, Jun 2008.
30
31 if ~exist('classname','var')
```

```
32      classname = class(A);
33  end
34
35  if exist('n','var')
36      C = bitcmp0(A, n);
37  else
38      C = bitcmp0(A);
39  end
40
41  if exist('bit','var')
42      err = biterrors(size(C), p, classname, bit);
43  else
44      err = biterrors(size(C), p, classname);
45  end
46
47  C = bitxor(C, err);
```

8.5 Inexact Bitwise AND

```
function [ C ] = bitand_inexact( A, B, p, classname, bit )
%bitand_inexact:  Calculates the bitwise AND of the input arguments A and
%B, similar to the standard bitand function, except that each bit has a
%random error probability equal to 1-p.
%
%Inputs:
%A, B:  (nonnegative integer arrays) Input arguments for the AND operator.
%    B must have the same dimensions as A.
%p:  (scalar) Probability of correctness of each bit within the output C.
%    0 <= p <= 1.
%classname:  (string) The class name of the output array C.
%bit:  (integer vector) Which bit positions can be inexact.  Position 1 is
%  lowest-order bit.  (optional) If bit is omitted, then all positions can
%  be inexact.
%
%Outputs:
%C:  (nonnegative integer array)  C = A AND B, subject to a bitwise random
%    error probability 1-p.  C has the same dimensions as A and B.
%
%Notes:
%If p=1, then C = A AND B and is error-free.
%If p=0, then C = A NAND B.
%If p=0.5, then C contains completely random data.
%
%Reference:
%L. N. B. Chakrapani and K. V. Palem, "A probabilistic Boolean logic and
%its meaning," Tech. Rep. TR-08-05, Rice University, Department of
%Computer Science, Jun 2008.

C = bitand(A, B);

if exist('classname','var')
    C = cast(C,classname);
```

```
34  else
35      classname = class(C);
36  end
37
38  if exist('bit','var')
39      err = biterrors(size(C), p, classname, bit);
40  else
41      err = biterrors(size(C), p, classname);
42  end
43
44  C = bitxor(C, err);
```

8.6 Inexact Bitwise OR

```
1  function [ C ] = bitor_inexact( A, B, p, classname, bit )
2  %Calculates the bitwise OR of the input arguments A and B,
       similar to the
3  %standard bitor function, except that each bit has a
       random error
4  %probability equal to 1-p.
5  %
6  %Inputs:
7  %A, B:  (nonnegative integer arrays) Input arguments for
       the OR operator.
8  %    B must have the same dimensions as A.
9  %p:  (scalar) Probability of correctness of each bit
       within the output C.
10 %    0 <= p <= 1.
11 %classname:  (string) The class name of the output array C
       .
12 %bit:  (integer vector) Which bit positions can be inexact
       .  Position 1 is
13 %  lowest-order bit.  (optional) If bit is omitted, then
       all positions can
14 %  be inexact.
15 %
16 %Outputs:
17 %C:  (nonnegative integer array)  C = A OR B, subject to a
       bitwise random
18 %    error probability 1-p.  C has the same dimensions as
       A and B.
19 %
20 %Notes:
21 %If p=1, then C = A OR B and is error-free.
22 %If p=0, then C = A NOR B.
23 %If p=0.5, then C contains completely random data.
24 %
25 %Reference:
26 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
27 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
28 %Computer Science, Jun 2008.
29
30 if nargin < 4
31     C = bitor(A, B);
32     classname = class(C);
33 else
```

```
34      C = zeros(size(A), classname);
35      C(:) = bitor(A, B);
36  end
37
38  if nargin >= 5
39      err = biterrors(size(C), p, classname, bit);
40  else
41      err = biterrors(size(C), p, classname);
42  end
43
44  C = bitxor(C, err);
```

8.7 Inexact Bitwise XOR

```
1  function [ C ] = bitxor_inexact( A, B, p, classname, bit )
2  %bitxor_inexact:  Calculates the bitwise XOR of the input
      arguments A and
3  %B, similar to the standard bitxor function, except that
      each bit has a
4  %random error probability equal to 1-p.
5  %
6  %Inputs:
7  %A, B:  (nonnegative integer arrays) Input arguments for
      the XOR operator.
8  %    B must have the same dimensions as A.
9  %p:  (scalar) Probability of correctness of each bit
      within the output C.
10 %    0 <= p <= 1.
11 %classname:  (string) The class name of the output array C
      .
12 %bit:  (integer vector) Which bit positions can be inexact
      .  Position 1 is
13 %  lowest-order bit.  (optional) If bit is omitted, then
      all positions can
14 %  be inexact.
15 %
16 %Outputs:
17 %C:  (nonnegative integer array)  C = A XOR B, subject to
      a bitwise random
18 %    error probability 1-p.  C has the same dimensions as
      A and B.
19 %
20 %Notes:
21 %If p=1, then C = A XOR B and is error-free.
22 %If p=0, then C = A XNOR B.
23 %If p=0.5, then C contains completely random data.
24 %
25 %Reference:
26 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
      Boolean logic and
27 %its meaning," Tech. Rep. TR-08-05, Rice University,
      Department of
28 %Computer Science, Jun 2008.
29
30 if nargin < 4
31     C = bitxor(A, B);
32     classname = class(C);
33 else
```

```
34      C = zeros(size(A), classname);
35      C(:) = bitxor(A, B);
36  end
37
38  if nargin >= 5
39      err = biterrors(size(C), p, classname, bit);
40  else
41      err = biterrors(size(C), p, classname);
42  end
43
44  C = bitxor(C, err);
```

8.8 Inexact 4-Input Bitwise AND

```
1  function [ E ] = bitand4_inexact( A, B, C, D, p, classname
       , bit )
2  %Calculates the bitwise (A AND B AND C AND D), similar to
       the bitand func-
3  %tion, except that each bit has a random error probability
        equal to 1-p.
4  %
5  %Inputs:
6  %A, B, C, D:  (nonnegative integer arrays) Input arguments
        for the AND4
7  %    function.  A, B, C, and D must all have the same
       dimensions.
8  %p:  (scalar) Probability of correctness of each bit
       within the output E.
9  %    0 <= p <= 1.
10 %classname:  (string) The class name of the output array E
       .
11 %bit:  (integer vector) Which bit positions can be inexact
       .  Position 1 is
12 %  lowest-order bit.  (optional) If bit is omitted, then
       all positions can
13 %  be inexact.
14 %
15 %Outputs:
16 %E:  (nonnegative integer array)  E = (A AND B AND C AND D
       ), subject to a
17 %    bitwise random error probability 1-p.  E has the same
        dimensions as A,
18 %    B, C, and D.
19 %
20 %Notes:
21 %If p=1, then E = (A AND B AND C AND D) and is error-free.
22 %If p=0, then E equals the bitwise complement of (A AND B
       AND C AND D).
23 %If p=0.5, then E contains completely random data.
24 %
25 %Reference:
26 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
       Boolean logic and
27 %its meaning," Tech. Rep. TR-08-05, Rice University,
       Department of
28 %Computer Science, Jun 2008.
29
30 E0 = bitand(B, bitand(C, D));
```

```
31  if nargin < 6
32      E = bitand(A, E0);
33      classname = class(D);
34  else
35      E = zeros(size(A), classname);
36      E(:) = bitand(A, E0);
37  end
38
39  if nargin >= 7
40      err = biterrors(size(E), p, classname, bit);
41  else
42      err = biterrors(size(E), p, classname);
43  end
44
45  E = bitxor(E, err);
```

Appendix I. Advanced Bitwise Functions

9.1 Unsigned to Signed Class Conversion

```
function [ b, sgn, signedclass ] = signed( a, classname )

if ~exist('classname','var')
    classname = regexprep(class(a),'uint','int');
end

if isa(a,'uint8')
    signedclass = false;
    n = 8;
    switch classname
        case {'int8','int16','int32','int64'}
            b = zeros(size(a),classname);
        otherwise
            error 'For 8-bit input, classname must be int8
               , int16, int32, or int64.'
    end
elseif isa(a,'uint16')
    signedclass = false;
    n = 16;
    switch classname
        case {'int16','int32','int64'}
            b = zeros(size(a),classname);
        otherwise
            error 'For 16-bit input, classname must be
               int16, int32, or int64.'
    end
elseif isa(a,'uint32')
    signedclass = false;
    n = 32;
    switch classname
        case {'int32','int64'}
            b = zeros(size(a),classname);
        otherwise
            error 'For 32-bit input, classname must int32
               or int64.'
    end
elseif isa(a,'uint64')
    signedclass = false;
    n = 64;
    switch classname
        case 'int64'
            b = zeros(size(a),classname);
```

```
40          otherwise
41              error 'For 64-bit input, classname must be
                    int64.'
42      end
43  else
44      signedclass = true;
45      b = a;
46      if nargout >= 2
47          sgn = (a < 0);
48      end
49  end
50
51  if ~signedclass
52      OxFFFF = intmax(class(a));
53      sgn = logical(bitget(a,n));
54      acomp = bitxor(a(sgn),OxFFFF);
55      m = (acomp == intmax(classname));
56      b1 = -cast(acomp + 1, classname);
57      b1(m) = intmin(classname);
58      b(sgn) = b1;
59      b(~sgn) = abs(a(~sgn));
60  end
```

9.2 Signed to Unsigned Class Conversion

```
function [ b, sgn, signedclass ] = unsigned( a, classname
    )

if ~exist('classname','var')
    classname = regexprep(class(a),'^int','uint');
end

if isa(a,'int8')
    signedclass = true;
    switch classname
        case {'uint8','uint16','uint32','uint64'}
            b = zeros(size(a),classname);
        otherwise
            error 'For 8-bit input, classname must be
               uint8, uint16, uint32, or uint64.'
    end
elseif isa(a,'int16')
    signedclass = true;
    switch classname
        case {'uint16','uint32','uint64'}
            b = zeros(size(a),classname);
        otherwise
            error 'For 16-bit input, classname must be
               uint16, uint32, or uint64.'
    end
elseif isa(a,'int32')
    signedclass = true;
    switch classname
        case {'uint32','uint64'}
            b = zeros(size(a),classname);
        otherwise
            error 'For 32-bit input, classname must be
               uint32 or uint64.'
    end
elseif isa(a,'int64')
    signedclass = true;
    switch classname
        case 'uint64'
            b = zeros(size(a),classname);
        otherwise
            error 'For 64-bit input, classname must be
               uint64.'
    end
else
```

```
40      signedclass = false;
41      b = a;
42      if nargout >= 2
43          sgn = (a < 0);
44      end
45  end
46
47  if signedclass
48      OxFFFF = intmax(classname);
49      sgn = (a < 0);
50      m = (a == intmin(class(a)));
51      b(sgn) = bitxor(cast(-a(sgn),classname),OxFFFF) + 1;
52      b(m) = cast(intmax(class(a)),classname) + 1;
53      b(~sgn) = a(~sgn);
54  end
```

9.3 Clear Upper Bits

```
function B = correct_upperbits( A, n )
%Takes an integer array A, stored as an m-bit signed or
%unsigned integer class (where m=8,16,32, or 64), and an
%integer n, and:
%    (1) For each A>=0, or if A is unsigned, clears the
%        uppermost m-n bits of A, and
%    (2) For each A<0, sets the uppermost m-n bits of A.
%
%The lower n bits remain unchanged.
%
%Inputs:
%    A:  (integer array) Input data.  Must be one of the
%integer classes.
%    n:  (integer) The number of lower bits of A which
   will
%remain unchanged.
%
%Output:
%    B:  (integer array) Output data, with upper bits set
   or
%cleared as described above.  B is the same class as A.

switch class(A)
    case 'int8'
        m = 8;    signedA = true;
    case 'uint8'
        m = 8;    signedA = false;
    case 'int16'
        m = 16;    signedA = true;
    case 'uint16'
        m = 16;    signedA = false;
    case 'int32'
        m = 32;    signedA = true;
    case 'uint32'
        m = 32;    signedA = false;
    case 'int64'
        m = 64;    signedA = true;
    case 'uint64'
        m = 64;    signedA = false;
    otherwise
        error 'Input␣A␣must␣be␣of␣the␣integer␣classes.'
end

if signedA
```

```
43      signA = logical(bitget(A,m));
44      OxFFFF = cast(-1,'like',A);
45      OxF000 = bitshift(OxFFFF,n);
46      Ox0FFF = bitcmp(OxF000);
47      B = zeros(size(A),'like',A);
48      if isscalar(n)
49          B(signA) = bitor(A(signA),OxF000);
50          B(~signA) = bitand(A(~signA),Ox0FFF);
51      else
52          B(signA) = bitor(A(signA),OxF000(signA));
53          B(~signA) = bitand(A(~signA),Ox0FFF(~signA));
54      end
55  else
56      OxFFFF = intmax(class(A));
57      OxF000 = bitshift(OxFFFF,n);
58      Ox0FFF = bitcmp(OxF000);
59      B = bitand(A,Ox0FFF);
60  end
```

9.4 Test if an N-Bit Number is Nonzero (Inexact)

```
function H = any_high_bits_inexact_PBL( X, n, p )
%Computes the bitwise OR of all bits in X.  If any bits in X are nonzero,
%then H is true; otherwise, H is false.  The algorithm uses a binary tree
%of 2-input OR gates to form an n-input OR gate.

if ~exist('p','var')
    p = 1;
end

k = [64,32,16,8,4,2,1];
k = k(k <= n);

OxFFFF = bitcmp0(zeros('like',X),n);
X = bitand(X,OxFFFF);

for k = k
    if mod(n,2)           % if n is odd, then OR the last two bits together
        X = bitset(X,2,or_inexact(bitget(X,2),bitget(X,1),p));
        X = bitshift(X,-1);
        n = n - 1;
        OxFFFF = bitcmp0(zeros('like',X),n);
        X = bitand(X,OxFFFF);
    end

    for i = 1 : k
        if (i+k) <= n
            X = bitset(X,i,or_inexact(bitget(X,i),bitget(X,i+k),p));
        end
    end

    n = k;
    OxFFFF = bitcmp0(zeros('like',X),n);
    X = bitand(X,OxFFFF);
end

H = logical(bitget(X,1));

end
```

9.5 Inexact Barrel Shifter

```
1  function [ C, stickybit, Cs ] = bitshifter_inexact_PBL( A,
       B, na, nb, p )
2  %bitshifter returns A shifted B bits to the left (
       equivalent
3  %to multiplying A by 2^B), similar to the Matlab bitshift
4  %function, except this function simulates a barrel shifter
5  %in a digital electronic circuit.  If B is positive, then
       A
6  %is shifted left. If B is negative, then A is shifted
       right.
7  %Any overflow or underflow bits are truncated.
8  %
9  %This barrel shifter is subject to random errors at each
10 %node in the circuit -- that is, every AND or OR gate has
       a
11 %random error probability equal to 1-p.
12 %
13 %Inputs:
14 %A:  (integer array) Number(s) to be shifted.
15 %B:  (integer array) Number of bit positions that A is to
       be
16 %    shifted.
17 %na:  (integer) Word size (number of bits) of A.
18 %nb:  (integer) Word size of B.
19 %p:  (scalar) Probability of correctness of each AND or OR
20 %    gate inside the barrel shifter.  0 <= p <= 1.
21 %
22 %Output:
23 %C:  (intger array) The result of shifting A to the left
       by
24 %    B bits.
25 %stickybit:  (logical array) The logical OR of all bits
       lost
26 %    due to truncation.
27 %Cs:  (integer array) For B<0, the stickybit is OR'd with
28 %    the least significant bit of C.  For B>0, the
       stickybit
29 %    is OR'd with the most significant bit of C.
30
31 if ~exist('p','var')
32     p = 1;
33 end
34
35 sticky = (nargout >= 2);
```

```
36
37  OxFFFF = cast(pow2a(na,'uint64') - 1, 'like', A);
38
39  S = sign(B);
40  B = abs(B);
41  L = (S > 0);      % left shift
42  R = ~L;           % right shift
43
44  C = bitand(A,OxFFFF);
45  bb = zeros(size(B),'like',A);
46  bb_ = bb;
47  ilsb = bb;
48
49  for i = 1 : nb
50      bb(:) = bitget(B(:),i);
51      bb_(:) = 1 - bb(:);
52      bb(:) = bb(:) * OxFFFF;
53      bb_(:) = bb_(:) * OxFFFF;
54      k = pow2(i-1);
55      C1 = bitshift0(C,k*S,na);
56      C1(R) = bitand_inexact(C1(R),bb(R),p,class(A),1:(na-k)
            );
57      C1(L) = bitand_inexact(C1(L),bb(L),p,class(A),(k+1):na
            );
58      C0 = bitand_inexact(C,bb_,p,class(A),1:na);
59
60      %%%%%%%%%% Compute sticky bit %%%%%%%%%%
61      if sticky
62          iR = min([na,pow2(i-1)]);
63  %         iL = max([0,na-pow2(i-1)]);
64          iL = na - iR;
65          ilsb(:) = pow2a(iR,'uint64') - 1;
66          ilsb(L) = bitshift(ilsb(L),iL);
67          C2 = bitand(C,ilsb);
68          C2(R) = bitand_inexact(C2(R),bb(R),p,class(A),1:iR
                );
69          C2(L) = bitand_inexact(C2(L),bb(L),p,class(A),(iL
                +1):na);
70          if i == 1
71              stickybit = logical(C2);
72          else
73              C2(L) = bitshift(C2(L),-iL);
74              C3 = any_high_bits_inexact_PBL(C2,iR,p);
75              stickybit = or_inexact(stickybit,C3,p);
76          end
77      end
```

```
78      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
79
80      C(R) = bitor_inexact(C0(R),C1(R),p,class(A),1:(na-k));
81      C(L) = bitor_inexact(C0(L),C1(L),p,class(A),(k+1):na);
82  end
83
84  if (nargout >= 3) && sticky             % merge sticky bit
       with final output
85      Cs = C;
86      R = (S < 0);      % right shift
87      sb0 = cast(stickybit,'like',C);
88      Cs(R) = bitor_inexact(Cs(R), sb0(R), p, class(C), 1);
89      Cs(L) = bitset(Cs(L), na, or_inexact(bitget(Cs(L),na),
            sb0(L), p));
90  end
```

9.6 Inexact Comparator

```
1  function [ AgtB, AltB, AeqB ] = comparator_inexact_PBL( n,
       A, B, p )
2  %comparator_inexact_PBL simulates an integer comparator
3  %using inexact digital logic, and returns logical values
       as
4  %follows:
5  %    AgtB = true if A>B, false otherwise,
6  %    AltB = true if A<B, false otherwise, and
7  %    AeqB = true if A=B, false otherwise.
8  %Each AND, OR, and NOT gate in the comparator is subject
       to
9  %a random error probability 1-p, consistent with the model
10 %of Probabilistic Boolean Logic.
11 %
12 %Inputs:
13 %n:  (integer) Number of bits processed by the comparator.
14 %A, B:  (integer arrays) Integers to be compared.
15 %p: (scalar) Probability of correctness of each gate
       within
16 %    the comparator.  0 <= p <= 1.
17 %
18 %Outputs:
19 %AgtB:  (logical array) True if A>B, false otherwise.
20 %AltB:  (logical array) True if A<B, false otherwise.
21 %AeqB:  (logical array) True if A=B, false otherwise.
22 %
23 %Reference:
24 %L. N. B. Chakrapani and K. V. Palem, "A probabilistic
25 %Boolean logic and its meaning," Tech. Rep. TR-08-05, Rice
26 %University, Department of Computer Science, Jun 2008.
27
28 if ~exist('p','var')
29     p = 1;
30 end
31
32 OxFFFF = bitcmp0(zeros('like',A),n);
33 A = bitand(A,OxFFFF);
34 B = bitand(B,OxFFFF);
35 Abar = bitxor_inexact(A,OxFFFF,p,class(A),1:n);
36 Bbar = bitxor_inexact(B,OxFFFF,p,class(B),1:n);
37 AbarB = bitand_inexact(Abar,B,p,class(A),1:n);
38 ABbar = bitand_inexact(A,Bbar,p,class(A),1:n);
39 AxnorB = bitnor_inexact(AbarB,ABbar,n,p,class(A),1:n);
40
```

```
41  C = false(numel(A),n);
42  C(:,n) = bitget(AxnorB(:),n);
43
44  AgtB = false(size(A));
45  AgtB(:) = bitget(ABbar(:),n);
46
47  for i = (n-1) : -1 : 1
48      C(:,i) = and_inexact(C(:,i+1),bitget(AxnorB(:),i),p);
49  end
50
51  for i = 1 : (n-1)
52      AgtB(:) = or_inexact(AgtB(:),and(C(:,i+1),bitget(ABbar
            (:),i)),p);
53  end
54
55  if nargout >= 2
56      AltB = false(size(A));
57      AltB(:) = bitget(AbarB(:),n);
58
59      for i = 1 : (n-1)
60          AltB(:) = or_inexact(AltB(:),and(C(:,i+1),bitget(
                AbarB(:),i)),p);
61      end
62
63      if nargout >= 3
64          AeqB = false(size(A));
65          AeqB(:) = C(:,1);
66      end
67  end
```

Appendix J. IEEE 754 Floating-Point Functions

10.1 Separate Floating-Point Number into its Components

```
function [ S, E, M, ne, nm, ee, mm ] = DecToIEEE754( X, fmt )
%BINARY16 (half precision)
%  Minimum value with graceful degradation:  6e-8
%  Minimum value with full precision:        6.104e-5
%  Maximum value:                            6.550e+4
%
%BINARY32 (single precision)
%  Minimum value with graceful degradation:  1.4e-45
%  Minimum value with full precision:        1.1754944e-38
%  Maximum value:                            3.4028235e+38
%
%BINARY64 (double precision)
%  Minimum value with graceful degradation:  4.9e-324
%  Minimum value with full precision:
   2.225073858507201e-308
%  Maximum value:
   1.797693134862316e+308
%
%BINARY128 (quadruple precision)
%  Minimum value with graceful degradation:
%  Minimum value with full precision:
%  Maximum value:
   1.1897314953572318e+4932

if ~exist('fmt','var')
    fmt = 'BINARY32';
end

switch upper(fmt)
    case 'BINARY16'
        ebias = 15;
        ne = 5;
        nm = 10;
        E = zeros(size(X),'uint8');
        M = zeros(size(X),'uint16');
        mnm = 1024;
        mna = 1023;
    case 'BINARY32'
        ebias = 127;
        ne = 8;
        nm = 23;
```

```
39          E = zeros(size(X),'uint8');
40          M = zeros(size(X),'uint32');
41          mnm = 8388608;
42          mna = 8388607;
43      case 'BINARY64'
44          ebias = 1023;
45          ne = 11;
46          nm = 52;
47          E = zeros(size(X),'uint16');
48          M = zeros(size(X),'uint64');
49          mnm = 4503599627370496;
50          mna = 4503599627370495;
51      case 'BINARY128'
52          ebias = 16383;
53          ne = 15;
54          nm = 112;
55          E = zeros(size(X),'uint16');
56          M = zeros(size(X),'double');
57          mnm = 5.1922968585348276e33;
58          mna = 5.1922968585348276e33;
59      otherwise
60          error 'fmt must be binary16, binary32, binary64, or binary128.'
61  end
62
63  S = (X < 0);
64  X = abs(X);
65
66  ee = floor(log2(X));
67  i = (ee <= -ebias);                 % graceful degradation toward zero
68  ee(i) = -ebias;
69  E(:) = ee + ebias;
70
71  mm = X ./ pow2(ee);
72
73  mm1 = mm;
74  mm1(~i) = mm1(~i) - 1;
75  M(~i) = mm1(~i) * mnm;
76  M(i) = 0.5 * mm1(i) * mnm;
77
78  c = (M >= mnm) & (~i);
79  E(c) = E(c) + 1;
80  M(c) = M(c) - mnm;
81
82  j = (isinf(X) | (ee > ebias));      % infinity
```

```
83  ee(j) = ebias + 1;
84  E(j) = ee(j) + ebias;
85  mm(j) = 0;
86  M(j) = 0;
87
88  k = isnan(X);                          % NaN
89  ee(k) = ebias + 1;
90  E(k) = ee(k) + ebias;
91  mm(k) = mna;
92  M(k) = mna;
93
94  z = (X == 0);                          % zero
95  ee(z) = 0;
96  mm(z) = 0;
97  E(z) = 0;
98  M(z) = 0;
```

10.2 Merge Components into a Floating-Point Number

```
function X = IEEE754toDec( S, E, M, fmt, infnan )
%BINARY16 (half precision)
%  Minimum value with graceful degradation:  6e-8
%  Minimum value with full precision:        6.104e-5
%  Maximum value:                            6.550e+4
%
%BINARY32 (single precision)
%  Minimum value with graceful degradation:  1.4e-45
%  Minimum value with full precision:        1.1754944e-38
%  Maximum value:                            3.4028235e+38
%
%BINARY64 (double precision)
%  Minimum value with graceful degradation:  4.9e-324
%  Minimum value with full precision:
   2.225073858507201e-308
%  Maximum value:
   1.797693134862316e+308
%
%BINARY128 (quadruple precision)
%  Minimum value with graceful degradation:
%  Minimum value with full precision:
%  Maximum value:
   1.1897314953572318e+4932

if ~exist('fmt','var')
    fmt = 'BINARY32';
end

if ~exist('infnan','var')
    infnan = true;
end

switch upper(fmt)
    case 'BINARY16'
        ebias = 15;
        einf = 31;
        mnm = 1024;
    case 'BINARY32'
        ebias = 127;
        einf = 255;
        mnm = 8388608;
    case 'BINARY64'
        ebias = 1023;
        einf = 2047;
```

```
42          mnm = 4503599627370496;
43      case 'BINARY128'
44          ebias = 16383;
45          einf = 32767;
46          mnm = 5.1922968585348276e33;
47      otherwise
48          error 'fmt must be binary16, binary32, binary64, or binary128.'
49  end
50
51  if (isinteger(E) || isinteger(M)) && all(E(:) >= 0)
52      ee = double(E) - ebias;
53      i = (E <= 0);                   % graceful degradation toward zero
54
55      mm1 = zeros(size(M));
56      mm1(~i) = double(M(~i)) / mnm;
57      mm1(i) = 2 * double(M(i)) / mnm;
58      mm = mm1;
59      mm(~i) = mm(~i) + 1;
60  else
61      ee = double(E);
62      mm = double(M);
63  end
64
65  X = pow2(ee) .* mm;
66
67  j = ((E >= einf) & (M == 0));       % infinity
68  k = ((E >= einf) & (M > 0));        % NaN
69  if infnan
70      X(j) = Inf;
71      X(k) = NaN;
72  end
73
74  z = ((E == 0) & (M == 0));          % zero
75  X(z) = 0;
76
77  S = (S ~= 0);
78  X(S) = -X(S);
79
80  switch upper(fmt)
81      case {'BINARY16','BINARY32'}
82          X = cast(X,'single');
83  end
```

Bibliography

1. Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr 1965.

2. Gordon E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 Int.*, volume 21, pages 11–13, Dec 1975.

3. Gordon E. Moore. Lithography and the future of Moore's law. In *Proc. SPIE vol. 2437*, pages 2–17, May 1995.

4. The MIT probabilistic computing project. [Online] Available: http://probcomp.csail.mit.edu, accessed 6 Jul 2015.

5. Krishna V. Palem. Energy aware computing through probabilistic switching: a study of limits. *IEEE Trans. Comput.*, 54(9):1123–1137, Sept 2005.

6. J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, October 1980.

7. L. Chakrapani and K. V. Palem. A probabilistic Boolean logic for energy efficient circuit and system design. In *2010 15th Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 628–635, Jan 2010.

8. B. E. S. Akgul, L. N. Chakrapani, P. Korkmaz, and K. V. Palem. Probabilistic CMOS technology: A survey and future directions. In *2006 IFIP Int. Conf. Very Large Scale Integration*, pages 1–6, Oct 2006.

9. Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *Proc. Conf. Design, Automation and Test Europe (DATE)*, DATE '06, pages 1110–1115, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

10. C. Mead. Neuromorphic electronic systems. *Proc. IEEE*, 78(10):1629–1636, Oct 1990.

11. Shimeng Yu, D. Kuzum, and H.-S. P. Wong. Design considerations of synaptic device for neuromorphic computing. In *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pages 1062–1065, June 2014.

12. Chi-Sang Poon and Kuan Zhou. Neuromorphic silicon neurons and large-scale neural networks: challenges and opportunities. *Frontiers in Neuroscience*, 5(108), 2011.

13. P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45 pJ per spike in 45 nm. In *Custom Integrated Circuits Conf. (CICC), 2011 IEEE*, pages 1–4, Sept 2011.

14. R. H. Dennard, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid-State Circuits*, 9(5):256–268, 1974.

15. Joel Hruska. Probabilistic computing: Imprecise chips save power, improve performance, Oct 2013. [Online] Available: http://www.extremetech.com/computing/168348-probabilistic-computing-imprecise-chips-save-power-improve-performance, accessed 6 Jul 2015.

16. M. Damiani and G. DeMicheli. Don't care set specifications in combinational and synchronous logic circuits. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 12(3):365–388, Mar 1993.

17. J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, Princeton, NJ, 1956.

18. Claude E. Shannon. Von Neumann's contributions to automata theory. In J. C. Oxtoby, B. J. Pettis, and G. B. Price, editors, *Bulletin of the American Mathematical Society, Vol. 64, No. 3, Part 2*, pages 123–129. American Mathematical Society, Providence, RI, May 1958.

19. S. Ulam. John Von Neumann 1903-1957. In J. C. Oxtoby, B. J. Pettis, and G. B. Price, editors, *Bulletin of the American Mathematical Society, Vol. 64, No. 3, Part 2*, pages 123–129. American Mathematical Society, Providence, RI, May 1958.

20. N. Pippenger. Developments in "the synthesis of reliable organisms from unreliable components". In James Glimm, John Impagliazzo, and Isadore Singer, editors, *Proc. Symp. Pure Mathematics, Vol. 50: The Legacy of John Von Neumann*, pages 311–324. American Mathematical Society, Providence, RI, 1990.

21. A. Lingamneni, C. Enz, J.-L. Nagel, K. Palem, and C. Piguet. Energy parsimonious circuit design through probabilistic pruning. In *Design, Automation and Test in Europe Conf and Exhibition (DATE), 2011*, pages 1–6, 2011. DOI: 10.1109/DATE.2011.5763130.

22. David A. Mindell. *Digital Apollo: Human and Machine in Spaceflight*. The MIT Press, Cambridge, MA, 2008.

23. Eldon C. Hall. A case history of the AGC integrated logic circuits, E-1880. Technical report, MIT Instrumentation Laboratory, Cambridge, MA, Dec 1965.

24. Technology horizons: A vision for air force science and technology 2010-30. Technical report, Air University Press, Maxwell AFB, AL, Sep 2011.

25. The USC signal and image processing institute (SIPI) image database. [Online] Available: http://sipi.usc.edu/database, accessed 24 Aug 2015.

26. Avinash Lingamneni, Christian Enz, Krishna Palem, and Christian Piguet. Parsimonious circuits for error-tolerant applications through probabilistic logic minimization. In *Proc PATMOS*, pages 204–213, 2011. DOI: 10.1007/978-3-642-24154-3_21.

27. Krishna V. Palem and Avinash Lingamneni. Ten years of building broken chips: The physics and engineering of inexact computing. *ACM Trans. Embedded Computing Syst.*, (Special Issue on Probabilistic Embedded Computing), 2012.

28. Krishna Palem and Avinash Lingamneni. What to do about the end of Moore's law, probably! In *Proc 49th Annual Design Automation Conf*, DAC '12, pages 924–929, New York, NY, USA, 2012. ACM. DOI: 10.1145/2228360.2228525.

29. Jaeyoon Kim, Sanghyeon Lee, J. Rubin, Moonkyung Kim, and Sandip Tiwari. Scale changes in electronics: Implications for nanostructure devices for logic and memory and beyond. *Solid-State Electron.*, 84:2–12, Jun 2013. DOI: 10.1016/j.sse.2013.02.031.

30. P. Korkmaz. *Probabilistic CMOS (PCMOS) in the Nanoelectronics Regime*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, Jul 2007.

31. Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *Proc. Conf. Design, Automation and Test Europe (DATE)*, DATE '10, pages 1560–1565, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

32. Lakshmi N. Chakrapani, Pinar Korkmaz, Bilge E. S. Akgul, and Krishna V. Palem. Probabilistic system-on-a-chip architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):29:1–29:28, Aug 2007.

33. J. George, B. Marr, B. E. S. Akgul, and K. V. Palem. Probabilistic arithmetic and energy efficient embedded signal processing. In *Proc. 2006 Int. Conf. Compilers, Architecture and Synthesis Embedded Syst.*, CASES '06, pages 158–168, New York, NY, USA, 2006. ACM.

34. Jaeyoon Kim and Sandip Tiwari. Inexact computing using probabilistic circuits: Ultra low-power digital processing. *ACM J. Emerging Technol. Comput. Syst.*, 10(2):16:1–16:23, Feb 2014.

35. Avinash Lingamneni, Kirthi Krishna Muntimadugu, Christian Enz, Richard M. Karp, Krishna V. Palem, and Christian Piguet. Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling. In *Proc 9th Conf Computing Frontiers*, CF '12, pages 3–12, Cagliari, Italy, 2012. ACM. DOI: 10.1145/2212908.2212912.

36. Ick-Sung Choi, Hyoung Kim, Shin-Il Lim, Sun-Young Hwang, Bhum-Cheol Lee, and Bong-Tae Kim. A kernel-based partitioning algorithm for low-power, low-area overhead circuit design using dont-care sets. *ETRI J.*, 24(6):473–476, Dec 2002.

37. T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley & Sons, New York, 1991.

38. L. N. B. Chakrapani and K. V. Palem. A probabilistic Boolean logic and its meaning. Technical Report TR-08-05, Rice University, Department of Computer Science, Jun 2008.

39. Alberto Leon-Garcia. *Probability, Statistics, and Random Processes for Electrical Engineering, 3rd ed.* Pearson Education, Upper Saddle River, NJ, 2009.

40. Neil H. E. Weste and David Money Harris. *CMOS VLSI Design, 4th ed.* Addison-Wesley, Boston, MA, 2011.

41. Huey Ling. High-speed binary adder. *IBM J. Research and Develop.*, 25(3):156–166, Mar 1981.

42. Pong-Fei Lu, G. A. Northrop, and K. Chiarot. A semi-custom design of branch address calculator in the IBM Power4 microprocessor. In *2005 Int. Symp. VLSI Design, Automation and Test (VLSI-TSA-DAT)*, pages 329–332, Apr 2005.

43. Robert M. Norton. The double exponential distribution: Using calculus to find a maximum likelihood estimator. *The American Statistician*, 38(2):135–136, May 1984.

44. Saralees Nadarajah. Exact distribution of the product of N Student's t RVs. *Methodology and Computing in Appl. Probability*, 14(4):997–1009, Dec 2012.

45. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. DOI: 10.1109/IEEESTD.2008.4610935.

46. V. Britanak. Discrete cosine and sine transforms. In K. R. Rao and P. C. Yip, editors, *The Transform and Data Compression Handbook*, chapter 4, pages 117–195. CRC Press, Boca Raton, FL, 2001.

47. K. R. Rao and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, and Applications.* Academic Press, San Diego, CA, 1990.

48. Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, Boston, 1995.

49. Eric Hamilton. *JPEG File Interchange Format, Version 1.02*. C-Cube Microsystems, Milpitas, CA, Sep 1992. [Online] Available: http://jpeg.org/public/jfif.pdf, accessed 21 Sep 2014.

50. M. L. Haque. A two-dimensional fast cosine transform. *IEEE Trans. Acoust., Speech, and Signal Process*, ASSP-33(6):1532–1538, Dec 1985. DOI: 10.1109/TASSP.1985.1164737.

51. M. Vetterli. Fast 2-d discrete cosine transform. In *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing, ICASSP '85*, volume 10, pages 1538–1541, Apr 1985. DOI: 10.1109/ICASSP.1985.1168211.

52. Bayesteh G. Kashef and Ali Habibi. Direct computation of higher-order dct coefficients from lower-order dct coefficients. volume 504, pages 425–431, 1984. DOI: 10.1117/12.944892.

53. Wen-Hsiung Chen, C. Smith, and S. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Trans. Commun.*, 25(9):1004–1009, Sep 1977. DOI: 10.1109/TCOM.1977.1093941.

54. Byeong Lee. A new algorithm to compute the discrete cosine transform. *IEEE Trans. Acoust., Speech, Signal Process.*, 32(6):1243–1245, Dec 1984. DOI: 10.1109/TASSP.1984.1164443.

55. C. Loeffler, A. Ligtenberg, and George S. Moschytz. Practical fast 1-d dct algorithms with 11 multiplications. In *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing, ICASSP-89*, volume 2, pages 988–991, May 1989. DOI: 10.1109/ICASSP.1989.266596.

56. F. A. Kamangar and K. R. Rao. Fast algorithms for the 2-d discrete cosine transform. *IEEE Trans. Comput.*, C-31(9):899–906, Sept 1982. DOI: 10.1109/TC.1982.1676108.

57. Nam-Ik Cho and San Uk Lee. Fast algorithm and implementation of 2-d discrete cosine transform. *IEEE Trans. Circuits Syst.*, 38(3):297–305, Mar 1991. DOI: 10.1109/31.101322.

58. Yukihiro Arai, Takeshi Agui, and Masayuki Nakajima. A fast DCT-SQ scheme for images. *Trans. IEICE*, E71(11):1095–1097, Nov 1988.

59. Peter Symes. *Video Compression*. McGraw-Hill, New York, 1998.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
24-12-2015	Dissertation	Sep 2012 - Dec 2015

4. TITLE AND SUBTITLE

Demonstration of Inexact Computing Implemented in the JPEG Compression Algorithm using Probabilistic Boolean Logic applied to CMOS Components

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Allen, Christopher I., Major, USAF

5d. PROJECT NUMBER

15G150

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-DS-15-D-001

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory, Sensors Directorate (AFRL/RYDI)
Attn: Bradley Paul
2241 Avionics Circle, Bldg 600
WPAFB OH 45433-7302
(937) 528-8706 (DSN: 798-8706) Bradley.Paul@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT Probabilistic computing offers potential improvements in energy, performance, and area compared with traditional digital design. This dissertation quantifies energy and energy-delay tradeoffs in digital adders, multipliers, and the JPEG image compression algorithm. This research shows that energy demand can be cut in half with noise-susceptible 16-bit Kogge-Stone adders that deviate from the correct value by an average of 3% in 14 nanometer CMOS FinFET technology, while the energy-delay product (EDP) is reduced by 38%. This is achieved by reducing the power supply voltage which drives the noisy transistors. If a 19% average error is allowed, the adders are 13 times more energy-efficient and the EDP is reduced by 35%. This research demonstrates that 92% of the color space transform and discrete cosine transform circuits within the JPEG algorithm can be built from inexact components, and still produce readable images. Given the case in which each binary logic gate has a 1% error probability, the color space transformation has an average pixel error of 5.4% and a 55% energy reduction compared to the error-free circuit, and the discrete cosine transformation has a 55% energy reduction with an average pixel error of 20%.

15. SUBJECT TERMS

Adders, FET Integrated Circuits, Image Compression, Inexact Computing, Integrated Circuit Noise, JPEG, Multipliers, Probabilistic Logic, Energy-Delay Product

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Derrick Langley (ENG)
U	U	U	UU	283	19b. TELEPHONE NUMBER *(include area code)* (310) 653-9113 Derrick.Langley@us.af.mil